

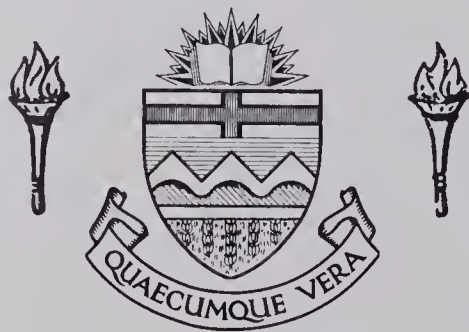
For Reference

NOT TO BE TAKEN FROM THIS ROOM

For Reference

NOT TO BE TAKEN FROM THIS ROOM

Ex LIBRIS
UNIVERSITATIS
ALBERTAENSIS



Regulations Regarding Theses and Dissertations

[illegible]

THE UNIVERSITY OF ALBERTA

COMPUTER PROBLEM-SOLVING

by

Richard David Peacocke

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE
OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

SPRING, 1969



Digitized by the Internet Archive
in 2019 with funding from
University of Alberta Libraries

<https://archive.org/details/Peacocke1969>

Thesis
R09
104

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled COMPUTER PROBLEM-SOLVING submitted by Richard David Peacocke in partial fulfilment of the requirements for the degree of Master of Science.

ABSTRACT

The thesis is concerned with heuristic solution of problems by digital computer. A particular program, the Graph Traverser (Doran and Michie 1966), can attempt to find a solution for any problem which can be formalised as that of finding a path from one specified node of a graph to another. The program has been implemented in an interactive system and emphasis placed on the user's control of the system. The results of different control schemes are presented and application of the program to some new problems is discussed.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the guidance given by his supervisor Dr. K.V. Wilson and the facilities and financial support from Dr. D.B. Scott, Chairman of the Department of Computing Science, University of Alberta. Thanks are also due to Mrs. B. Florchyk for her careful preparation of the final draft.

TABLE OF CONTENTS

	Page
CHAPTER I PROBLEM SOLVERS	1
CHAPTER II THE GRAPH TRAVERSER: ALGORITHM	23
CHAPTER III THE GRAPH TRAVERSER: OPERATIONAL FORM	31
CHAPTER IV EXPERIMENTS WITH THE GRAPH TRAVERSER	41
CHAPTER V CONCLUSIONS	60
REFERENCES	68
APPENDIX A GRAPH-THEORETIC TERMS	73
APPENDIX B IMPLEMENTATION OF GT IN A TIME-SHARING SYSTEM	76
APPENDIX C SOLUTION OF AN EIGHT-PUZZLE AND PROGRAM LISTINGS	78
APPENDIX D RANDOM EIGHT-PUZZLES USED (in Experiments 4.1, 4.2 & 4.3).	86

LIST OF TABLES

TABLE		Page
4.1	Performance of GT using three different retention functions	46
4.2	GT performance with three different flow thresholds	49
4.3	Search time for the solution of two Eight-puzzles using search algorithms S1 and S2	50
4.4	Solution of the N Queens problem using two different algorithms	54

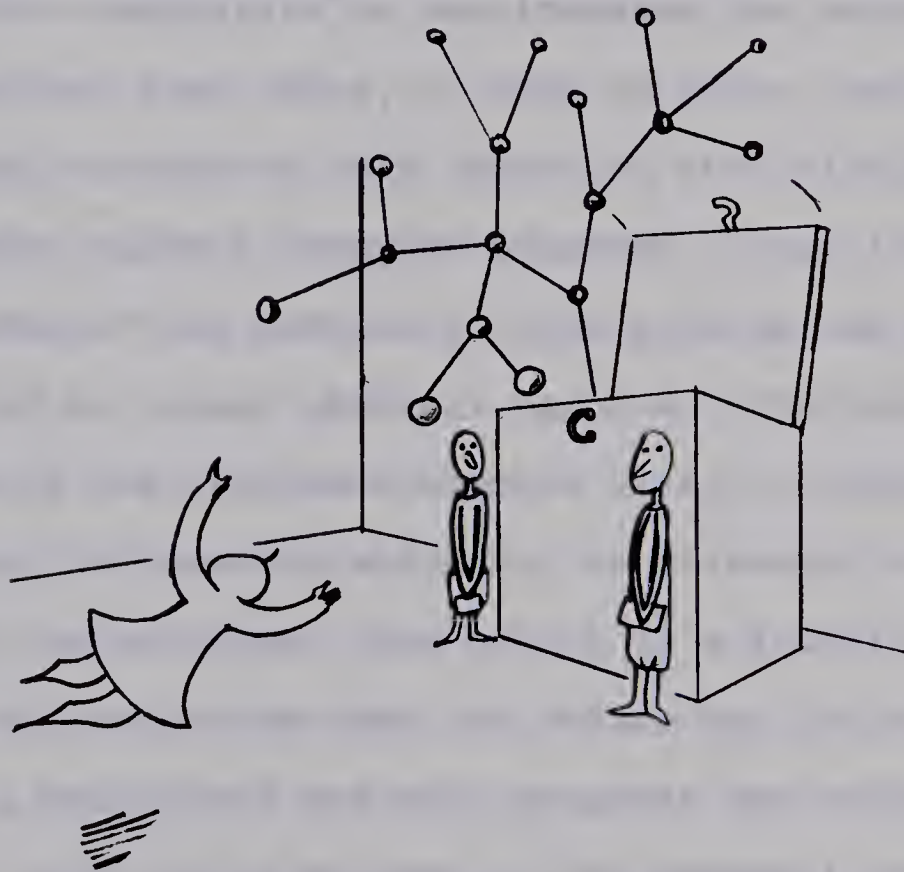
LIST OF FIGURES

FIGURES	Page
1.1 The components of a problem-solving system	6
1.2 'Backing' up through a simple game tree	9
1.3 An intelligence test problem of geometric analogy	11
1.4 The "chest of drawers" problem	14
2.1 The Graph Traverser in action	26
3.1 Components of the GT system	32
3.2 Two search trees	34
3.3 Construction of a single tree T by joining two others T_1 & T_2	37
3.4 A search tree showing the flow value for each leaf	38
3.5 PRUNE acting on a search tree	40
4.1 The retention functions	44
4.2 Paths traced to most promising nodes after a partial search	45
4.3 Effect of different flow thresholds on pruning	48
4.4 A solution of non-attacking queens on the 8x8 chessboard	51
4.5 Solution of the 4x4 Queens problem by backtracking.	55

THE ONLY SOLUTION

We shall have to evolve
problem-solvers galore --
since each problem they solve
creates ten problems more.

Piet Hein



CHAPTER I

PROBLEM SOLVERS

Introduction

How to win a game of chess is a 'problem'. How to persuade your bank manager to lend you money, how to solve a crossword puzzle, how to mend a car, these are more 'problems'. In all these situations the person attempting to find a solution has a system which he must manipulate until certain constraints or requirements are satisfied. A problem-solver must solve, or help to solve, problems.

The main concern of this thesis is the solution of problems using digital computer programs. Once loaded into the computer the problem-solving program can be thought of as an actual physical machine. The input to the machine is the problem statement itself, together with any auxiliary information which may be relevant to the problem and its solution. The output is a solution to the problem or an indication that the solver has failed, possibly with a note about how much progress was actually made.

Before work can be started on the external problem it must be converted into some internal representation. The methods which the problem solver has available will be applied to the internal problem and if a solution is found this will be converted back into external form. I shall have more to say about internal representations later in

this chapter but for the moment let us consider some strategies or methods of solution.

Algorithms

Associated with a problem is the set of all its candidate solutions. This set consists of all the possible manipulations to which the system may be subjected. Implicitly the problem statement also gives us a test to decide whether or not a particular element of this set is indeed a solution to the problem. The reason why problems are problems is that the actual solutions are generally scattered few and far between amongst the candidate solutions. The set of candidate solutions is not actually given as a list but can be found by generating the elements in some order. To each generation of an element we must attach a certain cost (time taken, memory space used, etc.) and some cost will be similarly associated with the test to see if an element is in fact a solution.

Some generators guarantee to find a solution sooner or later if the problem does indeed have one. Such a generator is called an 'algorithm' for that problem. But even when an algorithm is available it does not necessarily provide a useful means of attack in practice. The costs may be excessive.

Suppose we want to open a combination lock. A simple algorithm is to try all possible combinations, testing each

one in turn to see if the lock opens. We shall find the correct combination eventually but the time taken is large enough to make such a lock good protection against theft. I am not suggesting that algorithms never give practical methods of solution. They do sometimes; for example, the simplex algorithm of linear programming. However, for most of the problems to be considered in this work we need to use heuristic methods since no efficient enough algorithms are known.

Heuristics

The word 'heuristic' can be used both as an adjective and as a noun. It means 'serving to discover or find out' according to the Concise Oxford Dictionary which also tells us that 'heuristic method' means 'system of education under which the pupil is trained to find out things for himself'.

For any worthwhile problem blind search through all possibilities is too costly for practical use. And systems like chess, non-trivial parts of mathematics, and so on, are too complicated for complete analysis. We need to employ methods where the results of partial analysis can be used to make the search more efficient. Use is made of results along the way to guide the solution generator. Typically these methods are 'probably useful' rather than 'infallible', and they are called 'heuristic'.*

*In the literature 'heuristic' has often been regarded as the opposite to 'foolproof' but here it will not be used in this way. Imperfect methods are not necessarily heuristic, nor vice versa.

Progress, Heuristic Connection

If the search is to be guided at all we need two mechanisms. Firstly, some device to detect relative improvement. It must be able to judge whether the outcome of some particular trial solution is better or worse than another trial's outcome. Call this device a 'comparator' (Minsky 1961). Suppose the comparator-defined relation between trial outcomes is transitive. If A is better than B, and B is better than C then it follows that A is better than C. In this case we have defined 'progress' to our machine. But the comparator alone cannot help us do any better than straightforward exhaustive search. It tells us how we are progressing but gives no guidance about what to do next. We need to select new trials which are in some sense 'like', 'in the same direction as', or 'similar to' the trials which have given the best results so far (as judged by the comparator). To do this we need some structure on the space of candidate solutions telling us which points are heuristically related. Minsky (op.cit.) calls such a structure a 'heuristic connection' and emphasises that it will probably bear little resemblance to ordinary notions of distance and direction.

Planning

One way of solving a complicated problem is to break it up into series of less difficult 'subproblems'.

Suppose we were asked to integrate

$$\int \frac{x^4}{(1-x^2)^{5/2}} dx.$$

We might well try the substitution $y = \arcsin x$, transforming the original problem into the subproblem of integrating

$$\int \frac{\sin^4 y}{\cos^4 y} dy.$$

By trigonometric identities this could be transformed into $\int \tan^4 y dy$ or $\int \cot^4 y dy$, or by substituting $Z = \tan(y/2)$ we would get the subproblem

$$\int 32\{Z^4/(1+Z^2)(1-Z^2)^4\}dZ.$$

We would choose which of these subproblems to try and set to work on it in the same way. (We would hope eventually to reduce the integral to a standard form.) Setting up these related subproblems is called 'planning'.

A good plan should decrease our search time drastically. Suppose we had two safes each with a combination lock of ten dials and ten numbers on a dial. The first lock gives a click when any dial is turned to the correct number but the second one clicks only when all dials are correct. Using the algorithm of exhaustive search it would require, on the average, 5×10^9 trials to open the second lock. For the first lock we could set up the ten subproblems of getting each dial set correctly. We would solve these separately and opening the safe would take an

average of 50 trials. We achieve an enormous reduction of effort since the multiple searches add, rather than multiply, in the total search time.

We have discussed some aspects of a problem-solver in general terms (see Fig 1.1). Let us now take a brief look at some important heuristic programs which have been written to deal with specific tasks, see what they do and form a vague idea of how they do it.

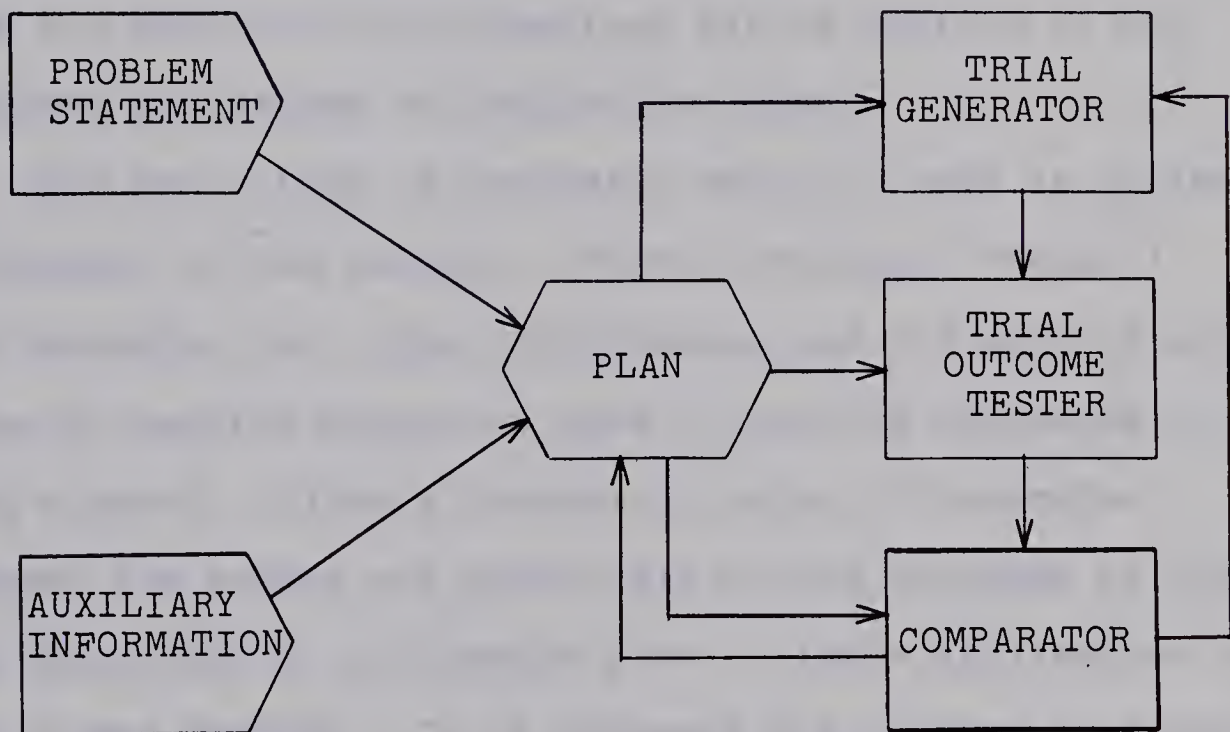


Fig 1.1 The components of a problem-solving system.

The Logic Theory Machine (LT)

This is a program (Newell & Simon 1956) written to prove theorems in elementary symbolic logic.* Theorem-proving, like game-playing, is a favorite topic for research into machine intelligence because the simplicity of the formal system allows concentration on the actual problem-solving processes rather than on detailed modelling of the task environment. The LT system is given as a set of five axioms and three rules of inference. These rules show how certain transformations can be applied to old theorems and axioms to produce new ones.

The major type of heuristic which LT uses is called a 'Method' by the authors. There are three 'Methods'. Each embodies the rules of inference and the mode of operation is centred round the idea of working backwards to find a proof. Given a theorem to prove, LT searches amongst the axioms and previously proved theorems to find one from which it can deduce T by a single application of one of the methods. If it succeeds the problem is solved. Or the search might fail but yield one or more propositions from which T could be deduced directly. If one of these

*LT was probably the first heuristic program fully realised on a computer and it is interesting to note that it gave rise to a list-processing language. This language, IPL (Newell & Shaw 1957), was used to handle the complex problems of memory allocation and hierarchical control of processing. It has since been refined in many different ways and list-processing is now a working tool in many different areas of Computing Science.

'subproblems' can be proved the main problem is solved. LT makes a preliminary attempt at solution and if this fails it adds the proposition to a subproblem list and works around to it later, in a recursive fashion.

Symbolic Automatic Integrator (SAINT, Slagle 1963)

SAINT was programmed to solve elementary symbolic integration problems at approximately the level of a good college freshman. If asked to integrate

$$\int x\sqrt{x^2+16}dx$$

it will, like a human solver, try various lines of attack and various substitutions in order to reach an integral of standard type.

The executive organisation of SAINT is like that of the Logic Theory Machine in using goal lists recursively. SAINT pays little attention to the problem of which line of attack to develop next but concentrates rather on how that attack is to be developed.

Samuel's Checker Player

The program (Samuel 1959) is written to perform well and little attempt is made to simulate human behaviour. It does perform remarkably well in fact and in 1962 defeated a former checkers champion of Connecticut.

As pointed out by Shannon (1950) two-person games like checkers are in principle finite. A best strategy

can be found by following all possible continuations of play to the end - if he goes there I can go there, there or there etc. This is equivalent to exhaustive search of our solution space. When we reach a terminal position we label it won, drawn or lost and eventually we 'back up' these values throughout the game tree (see Fig 1.2).

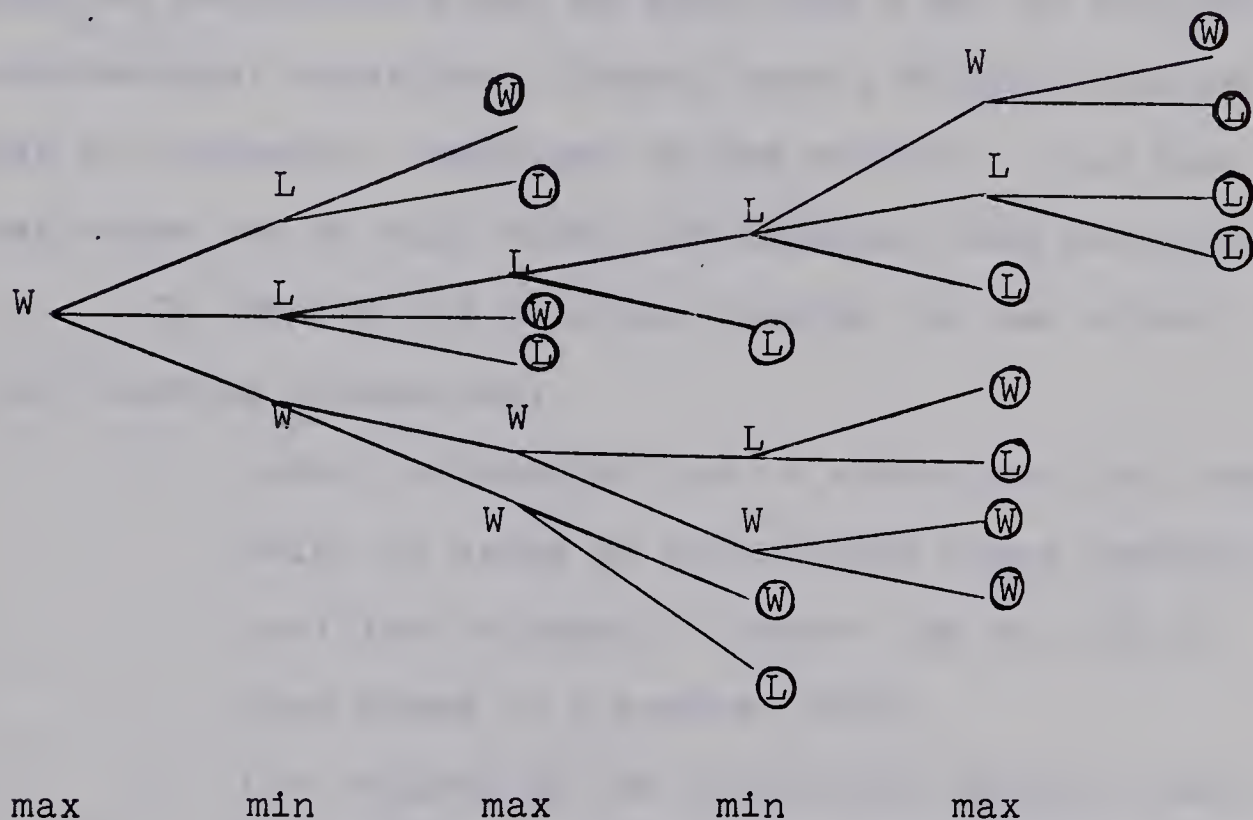


Fig 1.2 "Backing up" through a simple game tree. The terminal nodes have been labelled with the value of their outcomes to the first player, win or lose, shown ringed. Pre-terminal nodes are assigned values by the following rule: if the node represents the first player's choice label it with the maximum value of the nodes immediately following. If it is the opponent's choice label it with the minimum. (This assumes that the first player always wishes to maximise the outcome whilst the opponent wishes to minimise it.) The process is repeated until a value is established for the initial node, showing in this case a win for the first player.

In practice the game tree is generally colossal, for checkers perhaps 10^{40} positions. The tree must be pruned. We can limit the depth of exploration, restrict the number of alternatives explored from each position, or do both (for a more extensive discussion see Newell, Shaw & Simon 1958). If we are still going to 'back-up', but with an incomplete tree, we must find a way of evaluating non-terminal positions. Samuel uses a weighted sum of a set of 'property' functions of the position - how many men there are on each side, how advanced they are, etc.

To improve its play the program can use either of two learning mechanisms:

1. board information can be stored and the time which is saved by referencing these backed-up positions already in memory can be used to look ahead to a greater depth.
2. the weights in the evaluation function can be modified after each move in the light of discrepancies between the estimated value of the current board position and the values encountered during lookahead.

A Heuristic Program to Solve Geometrical Analogy Problems.

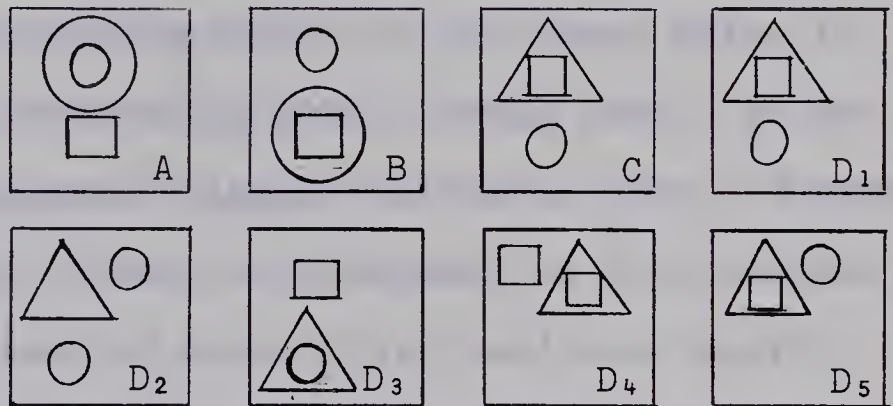
This program (Evans 1964) deals with 'intelligence test' problems and seems to be the only one written to perform this sort of analogical reasoning. The problem

is the recognition of analogies between geometric figures and the general format of the problem-question is:

'A is to B as C is to (D_1, D_2, D_3, D_4 or D_5)?' A, B, C and D are simple figures (see Fig 1.3).

Fig 1.3

An intelligence test problem of geometric analogy.



After finding topological and geometrical relations between the parts in each picture a hypothesis is developed about the relation of A to B. The correspondences between the parts of A and the parts of C are noted and the program searches for matchings of the A-B kind between the parts in C and each of the D-figures. The closest match gives us the required C-D relation and hence the D-figure to be chosen as the answer. The program is exceedingly complex and presently performs at about the level of a grade ten student.

There has been a good deal of interest in using natural language for communication between users and their programs. Earlier this interest gave rise to some question-answering programs (Green, Wolf, Chomsky & Laughery 1961; Lindsay 1963). More recently several problem solving programs have been written with some form of nat-

ural language input (Bobrow 1964; Kirsch 1964; Raphael 1964; Kuck & Krulee 1964).

Semantic Information Retriever (SIR, Raphael 1964).

SIR answers questions about a data base which it acquires through conversation with a human user. It can understand and manipulate simple statements like 'X owns Y', 'every J is a K', since the meanings (in this context, relational properties) of words like 'own' and 'every' have been given to it. Each variable could be replaced by a noun so a typical protocol is:

USER: 'The boy is to the left of the chair'

SIR: 'I understand'

USER: 'The chair is to the left of the table'

SIR: 'I understand'

USER: 'Is the table red?'

SIR: 'Insufficient information'

USER: 'Is the boy to the left of the table?'

SIR: 'Yes'

SIR uses a network of nodes with each one representing one of the objects SIR can discuss e.g. 'boy'. The relationships are represented by labelled links from one object to another. In this way the network actually forms a model of the situation.

STUDENT

This program (Bobrow 1964) attempts to solve simple story problems of high school algebra. Bobrow's main concern was to provide the computer with the ability to read the informal verbal statement of a problem and to derive from it the set of equations needed for solution. This derivation is the tricky part, for students as well as for the computer. The algebra itself is trivial. A typical problem and solution :

MARY IS TWICE AS OLD AS ANN
WAS WHEN MARY WAS AS OLD
AS ANN IS NOW. IF MARY IS
24 YEARS OLD, HOW OLD IS ANN?

ANN'S AGE IS 18.

The machine reads the problem statement and attempts to rewrite it as a number of simple sentences. It tries to convert each sentence into an equation and then attempts to solve the whole set of equations. The required answer is then converted back into a simple English sentence. Ambiguous or insufficient information is handled in a 'reasonable' way. To help problem interpretation and solution there is a library consisting of a dictionary, a variety of factual statements and some specialised problem solving subprograms. The whole system is so loosely organised that new information which may be needed can be added anywhere in the dictionary. In the preceding six heuristic programs and in others throughout the literature

we see some patterns crystallising out. We spend the next few sections looking at these patterns.

Problem Representation

A problem of major importance for all these problem-solvers is the internal representation the solver is to use. Choosing a good one may make all the difference between solving and not solving the problem. To take an example, suppose we have m indistinguishable objects to be placed in the n drawers of a chest. In how many ways can this be done?

We might try the enumeration approach, m in the first drawer, none in the rest, $(m-1)$ in the first, one in the second, $(m-1)$ in the first, one in the third, and so on. We would rapidly find that this was an impossibly difficult way of doing things. Imagine a different representation. Instead of a string of n numbers we have the m objects in a line, with two fixed partitions, one at each end. To construct n cells we need to insert $(n-1)$ extra partitions between the two fixed ones (see Fig 1.4).

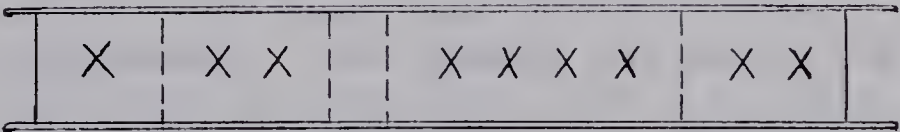


Fig 1.4 The "chest of drawers" problem.
The case $m=9$, $n=5$ is shown.

So between these end positions we have $(m+n-1)$ places which can be occupied by either an object or a partition. The total number of ways is then

$$\binom{m+n-1}{m} = (m+n-1)!/m! (n-1)!$$

Choosing a new representation is the same as looking at the problem in a different way. Amarel (1966) believes that the process of choosing and shaping appropriate representations for problem-solving is the essence of the behaviour in humans that we call 'creative'. I think it is an important component at least. A new paper by the same author (Amarel 1968) gives a formulation and extended discussion about the problem of choosing a representation, the first to do so in a general manner.

We can distinguish between two main types of internal representation. In one type we have a description of the problem and in the other a more direct representation of the situation viz. a model. For example, STUDENT sets up a description of the problem while SIR on the other hand constructs a model. When using a description it is easier to represent imperfect or incomplete information. But models can carry implicitly information that must be expressed if a description is used. One may have to state and use this information repeatedly in inferences when working with a descriptive system. I do not wish to argue for or against either representation

but merely remark on the difference between them.

Semantic Models

As an aid to setting up subgoals for a problem we may wish to interpret our internal representation. Suppose we wish to prove a theorem in plane Euclidean geometry. Imagine we have set up a chain of lemmas to help us solve the problem. We want to know if these lemmas really are true or else we shall be wasting effort in attempting to link them. We could construct the analytical geometry models of a few drawings and test the lemmas on these, effectively by actual measurement. If the drawings were well chosen the results would give a very reliable indication of the truth or falsity of the lemmas. Although of course they could not provide a proof they might provide a counter-example.

It may be possible to interpret the internal representation directly or one may need to set up a dual semantic model for this purpose. The geometry machine of Gelernter and Rochester (1958; 1959) uses such a model with excellent results. LT also uses interpretation, in the form of a 'strong non-provability test' to throw out false propositions.

Operator and State Selection

Work in heuristic problem-solving has tended to

cluster around two basic approaches. One approach is based on state evaluation. All the states of the system which can be reached from the current state in a small number of moves are generated. Some form of selection is applied to this new set of states to choose which one is to be used as the next point of generation. Most game-playing programs have used this approach (for review, see Michie 1966). The other way of doing things is to apply selection to the set of operators used to transform one state into another. In the studies of Newell, Shaw and Simon (1960) the operators are arranged in a priority sequence. Each operator is tested in turn to see

(i) if it is applicable to the current state of the system

(ii) if it helps to achieve any of the goals or subgoals of the problem.

I use the terminology of Doran and Michie (1966) and call these two approaches state and operator-selection respectively. Samuel's checker-player uses state selection whilst SAINT for example uses operator-selection.

Matching

In the review of Newell and Ernst (1965) they note that '.....a basic technique, matching two forms to determine appropriate substitutions for variables, which accounts for much of the success in the early heuristic

programs, continues to be an important technique in recent programs.' If we wish to produce an object X_d which is contained once in the set of candidate solutions $X_1, X_2, \dots, X_n$ and the order of generation of this set is random with respect to X_d , then we will need approximately $n/2$ productions and $n/2$ identity tests. But if we can make the different X 's a function of some variable y , by matching $X(y)$ to X_d we can determine y directly. The cost of the process no longer depends on n and this would seem to account for the success of the technique.

Selection of Subproblems

In the midst of solving a problem we generally have a list of subproblems which we are also working on. We also have available some methods or transformations to be used on the system. We cannot use all of the methods to achieve all of the subproblems all at the same time. Which subproblem do we attack and which method do we use? The question is posed in such general terms that it is impossible to answer but even in a specific problem-solving environment any answer at all is unlikely to be straightforward.

All we shall do here is mention two of the criteria that may be called on for selection of subproblems and indicate a crude classification of methods. For each subproblem we could estimate at least two things; the import-

ance of achieving the subgoal i.e. how far it takes us towards solving the central problem bearing in mind the way it may be linked to other subgoals; and the resources needed to solve the subproblem, how much time, memory space, external storage etc. is likely to be used up before it is achieved.

Then some scheduling algorithm would be necessary. The more important the subgoal, the more effort or resources we would be prepared to spend on it, but apart from this we can say little about the algorithm in general.

Methods of Attack

In SAINT there is a tripartite division of methods into standard forms, algorithm-like transformations and heuristic transformations. For a certain set of integrals we know the answer, eg. $\int c^v dv = c^v / (\log_e c)$. An integral is of standard form if we can transform it into one of this set by a simple substitution. Thus $\int 2^x dx$ is a standard form with solution $2^x / (\log_e 2)$.

The Logic Theory Machine uses simple substitutions as well and in these two programs we see instances of an 'immediately achieve' procedure. Matching is used in both to see if a substitution is applicable and which one will work. It eliminates enough trial and error in these substitutions and replacements to turn these programs into successful problem-solvers. The second type of method is

an algorithm-like transformation. When applicable to a subproblem it is almost always appropriate. We use another example from SAINT, the transformation of decomposition:

$$\int \sum g_i(v) dv = \sum \int g_i(v) dv$$

Finally we have heuristic transformations. Even when applicable it is quite possible that a heuristic transformation is not appropriate. It may take us no closer to a solution or there may be a better transformation. To complete the trio of examples from SAINT we have 'substitution for a subexpression whose derivative divides the integrand'. The 'Methods' of the Logic Theory Machine are all heuristic transformations.

A division of methods as outlined above occurs again and again in the heuristic programs for problem-solving. It seems to be a useful categorisation, although a somewhat obvious one. For a more comprehensive study see the review of Minsky (1961).

Generality

Early problem solving programs concentrated on difficult tasks like playing checkers or chess, proving theorems and the like. They were each written with a specific problem or family of problems in mind. Recently there has been a shift from difficulty towards generality of tasks. Now one is more concerned with making the solver accept problem statements in a general language (eg. SIR,

STUDENT) and work with a wider range of problems. This means that the internal representation must be more general and as a consequence the problems dealt with are less difficult. Indeed there seems to be a natural law, akin to some of Parkinson's Laws, that power and generality do not go hand in hand.

In the preceding sections I have tried to extract some of the underlying mechanisms of a few successful problem-solvers. A significant core of techniques has appeared and is continuing to grow. The minimaxing of Newell, Shaw and Simon is a classical example (although it was probably conceived first by Turing (1953)). Other interesting works are 'minimum cost paths' (Nilsson 1968), 'non-deterministic algorithms' (Floyd 1967), 'the alpha-beta procedure' (in Samuel 1959)) and 'backtrack programming' (Golomb and Baumert 1965). Some of these are welded together in MULTIPLE (Slagle and Bursky 1968).

The General Problem Solver

The GPS program (General Program Solver) of Newell, Shaw and Simon (1959) is well known. Not only did its authors manage to build a sophisticated problem solver but they also succeeded in casting some light into the murky area of human problem-solving.

GPS deals with a task environment consisting of 'objects' which can be transformed by various 'operators';

it detects 'differences' between objects; and it organises its information about the environment into 'goals'. There are three types of goal:

- (1) Transform one object into another
- (2) Reduce the difference between one object and another
- (3) Apply an operator to an object.

Objects and operators are given by the task; differences are something GPS brings to the problem. When presented with a goal the program seeks to achieve it or reduce it to a set of subsidiary goals. These subgoals form a tree and each one is taken and treated similarly. (A number of tests have to be applied to keep the tree from expanding too wildly).

In GPS a deliberate effort was made to separate the 'general' part of the program, the means-end-analysis, from the problem-specific part, the 'task environment'. By introducing appropriate task environments GPS can be applied to a variety of different problems eg. solving trigonometric identities, proving theorems of symbolic logic and compiling computer programs (Simon 1963). The problems are different but GPS works on each internal representation using the same strategy. The same is true of the Graph Traverser program and it is to this program that we turn in the next chapter.

CHAPTER II

THE GRAPH TRAVERSER: ALGORITHM

Introduction

The Graph Traverser is a problem-solving program for a digital computer. In this chapter we describe the algorithm and the types of problem to which it may be applied. In Chapter III the author's own modifications to the algorithm are explained and an implemented version of the program is outlined. Two headings describe most of the work in this thesis.

- (1) investigation of different methods to decide what information is to be retained and what is to be discarded in the course of problem-solving, and
- (2) solution of a typical allocation problem by GT.

In Chapter IV we describe some applications of GT and their results on various problems and present conclusions in Chapter V.

The original program is described in detail in Doran and Michie (1966). Doran (1967) presents some results from first applications of the problem-solver and discusses future developments. Michie (1967) sets out an evolutionary scheme for improvement of the simple Graph Traverser, and Doran (1968) describes application to the Travelling Salesman problem using the algorithm in a modified form.

GT will seek a solution to any problem capable of being formalised in the following way: a set of 'states' is given with a set of 'operators' connecting them. From an initial state (which may or may not be given) a path is to be found which leads to a 'goal' state. The goal state may be specified in full as part of the problem statement itself or only stated in terms of some defining property (eg. cost in an allocation problem). The path will be a sequence of states or the operators connecting them (cf. formulation in Newell and Ernst 1965). Minsky (1961) remarks, 'Almost any problem can be converted into a problem of finding a chain between two terminal expressions in some formal system'.

The Graph Traverser Algorithm

The formal representation of the problem and the strategy of GT can be conveniently described using the language of graph theory (see any standard text, Berge 1962, Busaacker and Saaty 1965, etc.). Appendix A presents informal definitions of some basic terms.

The problem is that of finding a path in a problem graph from a node labelled 'start' to a node labelled 'goal'. If we could see the entire graph at once the problem would be simple but this is not generally the case. The graph is defined as a set of distinct matrices corresponding to the nodes along with a function DEVELOP which

can be applied to any particular node and lists all its immediate descendants (in matrix form). Thus the graph definition is purely local and information about the entire structure is gained by probing in various directions.

The strategy of the search is based on an evaluation function which estimates the distance of any node from the goal. The function EVALUATE has to be written by the user for each particular problem and this is certainly a non-trivial task. So for each problem the user must write two functions (subroutines); DEVELOP, giving the 'rule-book' of allowable transformations, and EVALUATE which measures the apparent 'promise' of a node for further investigation. If EVALUATE produces a small value for some node it is worthwhile investigating that node further since it is probably near the goal. This investigation will take the form of finding all the node's immediate descendants by applying the function DEVELOP.

The search proceeds iteratively. At the beginning of an iteration the program has a list of the nodes so far produced by applications of DEVELOP. For each node the following information is stored;

- (i) the value produced by the function EVALUATE,
- (ii) a flag showing whether or not the node has been developed, and
- (iii) a pointer to the parent node.

The undeveloped node with the smallest value is

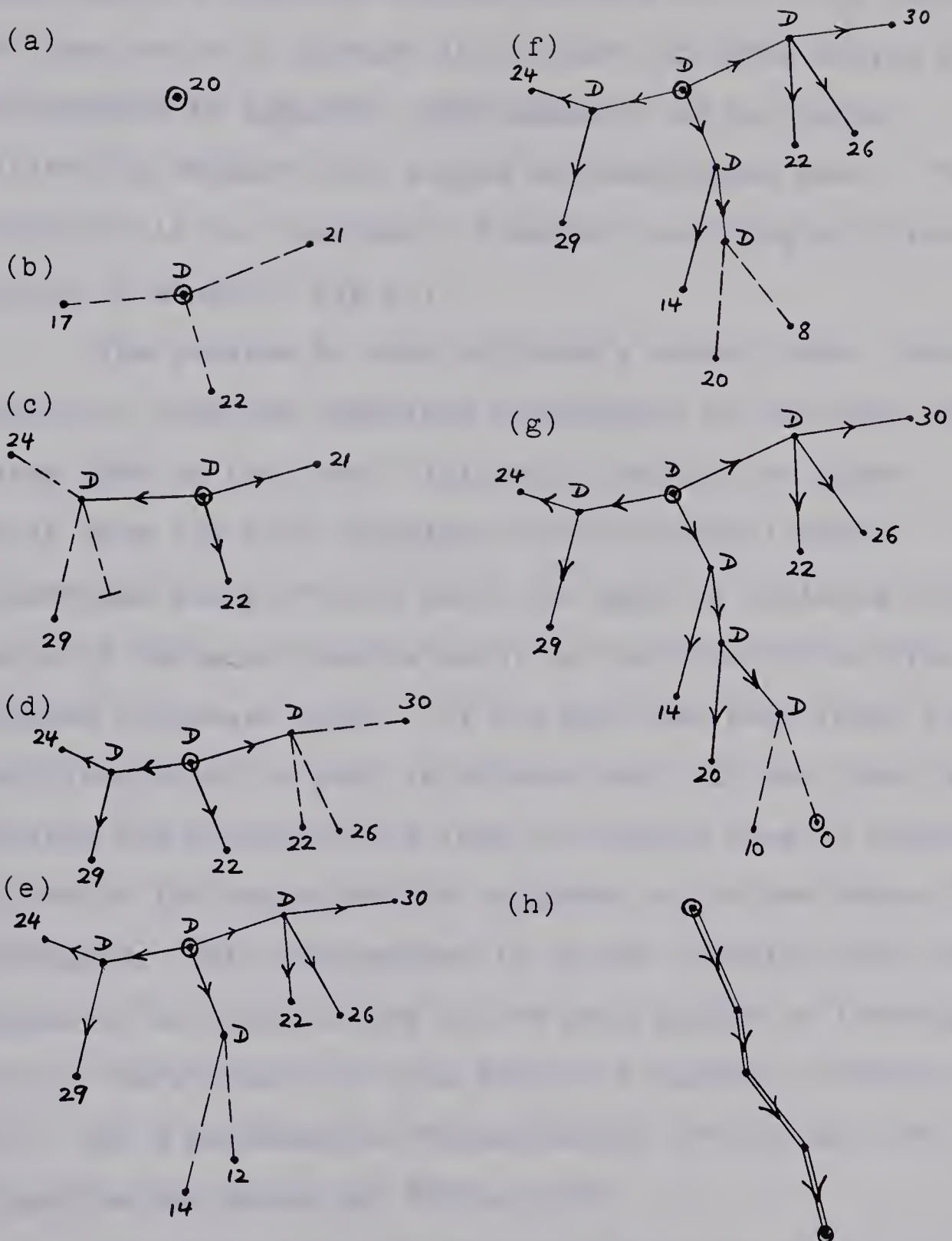


Fig 2.1 The Graph Traverser in action. Successive diagrams show a search tree being enlarged by successive developments until the goal is located and a solution path determined. The start and goal nodes are ringed, the symbol 'D' indicates a developed node, and the figures are the values assigned to nodes by the program's (imperfect) evaluation function. (h) shows the path solution. (Reproduced from Doran and Michie (1966)).

found and the function DEVELOP applied to it. Any descendant node which is already in the list of nodes stored by the program is ignored. The remainder of the nodes (listed by DEVELOP) are stored as undeveloped ones. The iteration is now complete. A search involving six iterations is shown in Fig 2.1.

The program is thus building a search tree. Each iteration adds the immediate descendants of the most promising node in the tree. Initially the list of nodes which form the tree contains only the 'start' node. Iterations are performed until the goal is achieved (the value of the goal node is zero) or the size of the tree reaches a pre-set limit. If the goal has been found the path from start to goal is printed out. If the tree size reaches the pre-set limit then the search tree is replaced by one of its own connected subgraphs and a new search is initiated. This replacement is called 'pruning' and the scheme to be used was one of the main topics of investigation. More detail on this subject is given in Chapter III. For a mathematical formalisation of the basic GT algorithm see Doran and Michie 1966.

Note that the program is not attempting to find the shortest path from start to goal. Maximum economy of the search is the criterion rather than elegance or shortness of the solution produced. In fact the program stops as soon as the first satisfactory solution is found and

no attempt is made to improve on this solution.

Application of the Graph Traverser

For each different problem we need to:

- (i) find a representation for the problem-states as matrices,
- (ii) write the function DEVELOP which lists all the immediate descendants of a problem-state, and
- (iii) write the function EVALUATE which estimates the distance of a problem-state from the goal.

In this way we supply the specific task environment and once this is done we apply GT to the representation we have produced. In this fashion the Graph Traverser can be applied to a variety of different problems just as the General Problem Solver can. In Chapter I we noted that a problem-solving program must repeatedly decide which sub-problem to attack next and which operators or transformations to apply. GT applies all possible operators (embedded in the DEVELOP function) to whichever state it chooses to work on next. This choice of state is completely dependent upon the evaluation function. If the values computed by this function are good predictions of the actual distances from the nodes to the goal then the search will be efficient. If the function is constant it conveys no information and the strategy is reduced to a systematic search of the problem graph working from the start node

outwards. The importance of the evaluation function should be clear by now and it seems worthwhile to repeat that this function, like DEVELOP, is written by the user for the particular application. Finding a suitable function EVALUATE is quite a problem in itself.

Initially an attempt was made to discover means whereby the evaluation function could improve itself. This would be a form of learning. The work met with little success and will be briefly dealt with in Chapter V.

Types of Problem

We can recognise two main problem classes; 'path' and 'property' searches. In a path search a path is sought which connects the start and goal states. Recall that the path may be asked for as a sequence of connected states or the corresponding operators may be required. Solution in the second form is of a slightly stronger kind since we can derive the states from the operators but the converse is not necessarily true. The start state is given in the problem statement and the goal state is probably given explicitly as well.

In a property search we look for a state satisfying certain constraints. Once any such state is found the problem is solved. Obviously the goal state is not given explicitly. The whole structure of the problem graph is no longer inherent in the problem itself. It is a matter

of our own definition and we also need to find a suitable initial state from which to start the Graph Traverser. The representation of problem states and the definition of suitable operators is of great importance.

CHAPTER III

THE GRAPH TRAVERSER: OPERATIONAL FORM

Introduction: Program Structure

The Graph Traverser has been programmed for the IBM System 360 in such a way as to interact with the user. It no longer operates in isolation. This conversational form seems to be the way in which most problem-solving programs will be written, at least in the near future. The user helps the program where it cannot cope by itself and vice versa. For an outline of the APL\360 system under which GT operates see Appendix B. In this chapter we describe the Graph Traverser program and explain the modifications that have been made to the original algorithm.

There are three main routines, or functions as they are known in APL. These are SEARCH, FIGS and PRUNE.

- (i) SEARCH grows the tree until it contains the goal node or until the resources of space or time are exhausted. We call such a growth of a search tree a 'partial search'.
- (ii) FIGS displays to the user facts and statistics about the tree, how many nodes it contains, its height, average value of its undeveloped nodes, etc.
- (iii) PRUNE reduces the full tree to a smaller structure with which to initiate the next search.

Several other functions, including DEVELOP and EVALUATE of course, are called by these three main functions.

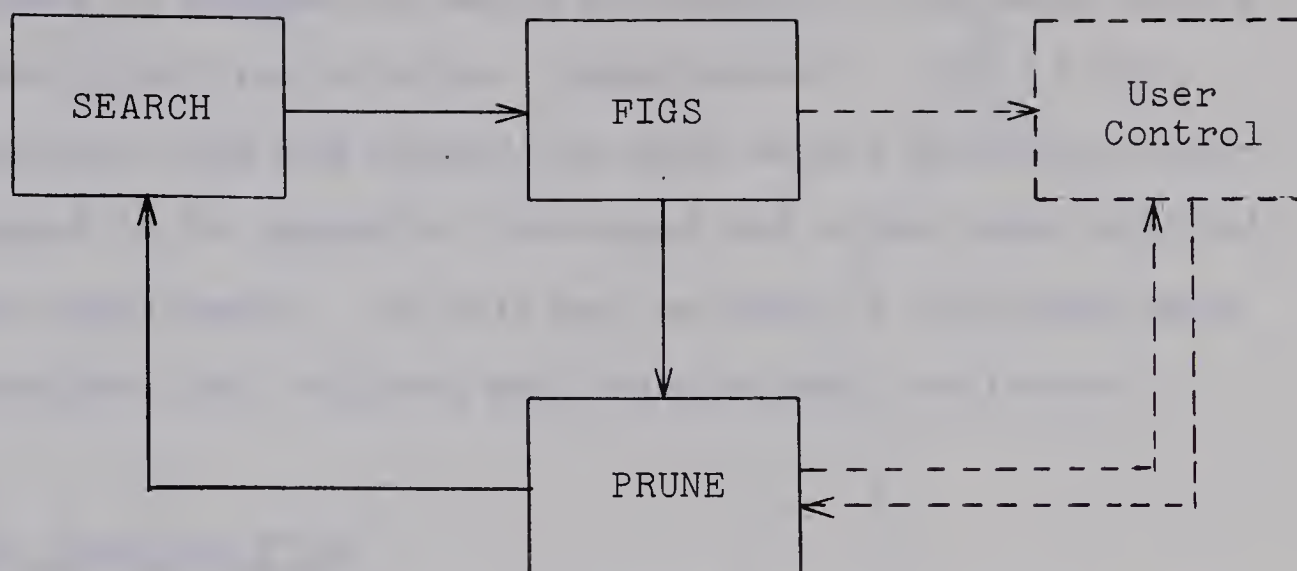


Fig 3.1 Components of the GT system. The dotted lines show the interaction with the user.

The Function SEARCH

Doran and Michie's iterative scheme for growing the search tree has been modified resulting in a significant increase in speed. In their method (as described in Chapter II) a node is selected* for development at each iteration. This node is fully developed and each one-arc descendant is added to the tree if and only if it is not already contained in the tree. Testing to check for the previous occurrence of a node is time-consuming. Instead we select the node as before and, providing that an identical node has not been developed previously, develop it fully. Each immediate descendant is added to the tree

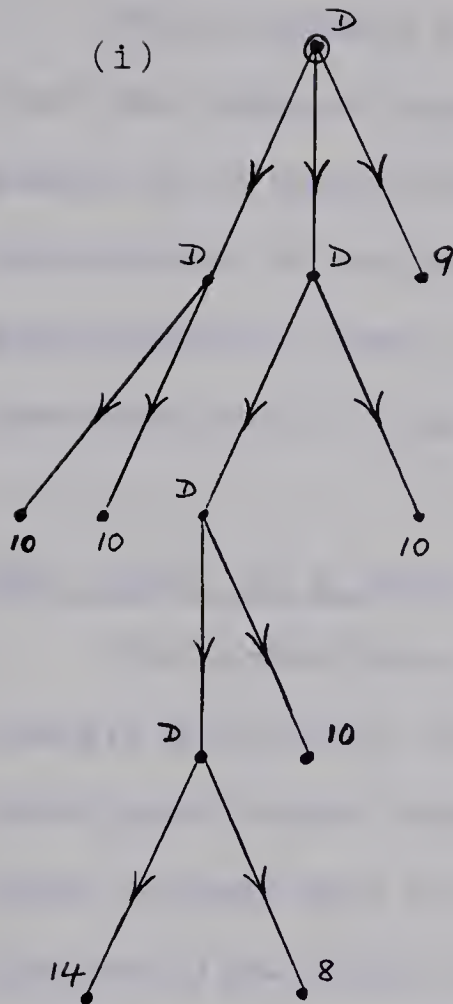
* If there is a tie for development (several nodes with the same value) a random choice is made from the nodes closest to the root of the tree.

unless it happens to match the parent of the node we are developing (ie. it's own 'grandparent'). But if the selected node was indeed the same as one previously developed it is marked as developed and a new node selected for development. In this way no node is developed more than once but the tree may contain some duplicates.

The Function FIGS

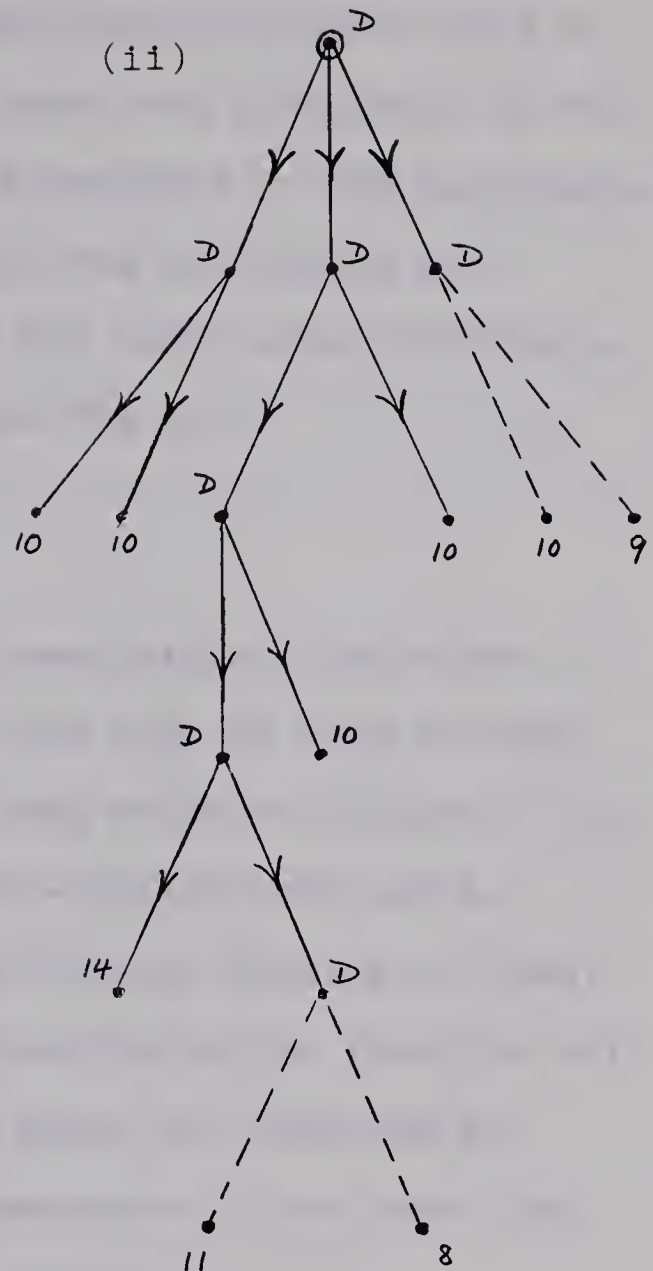
When a search tree has been grown the user needs some information about its structure. He wants to know which parts of the tree to discard before starting the next search and his decision will depend on the performance of the search just completed. The function FIGS supplies relevant information to help him make this choice. For the actual output items see Appendix C.

One of these items viz. 'penetrance' will need some explanation. Doran and Michie (1966) introduce it as a measure of efficiency of the search and define it as the fraction of the total number of nodes developed which are incorporated into the actual path found during the search. This is the ratio P/D where P is the length of the path produced (from root to the minimum-valued leaf) and D is the total number of nodes developed. In place of this we use the ratio H/E where H is the height of the search tree and E is the number of nodes it contains which are not leaves.



$$P/D = 4/5$$

$$H/E = 4/5$$



$$P/D = 2/7$$

$$H/E = 5/7$$

Fig 3.2 Two search trees. (ii) is obtained from (i) by making two developments more. The two penetrance figures for each tree are shown for comparison. Note how rapidly P/D has dropped. If the penetrance is high the tree is 'elongated', if it is low then the tree is 'bushy'.

The slightly different definition has been used so that the measure hopefully reflects the efficiency of the search as a whole without undue emphasis on the particular path chosen at the end. Usually the two ratios are approximately equal, but there are cases when P/D fluctuates violently. See for example Fig 3.2.

The 'Flow' of a Node

The penetrance gives us some guidance about the overall efficiency of a search but then we need to know which particular nodes to keep and which to discard. Our first thought may be to keep low-valued nodes and to discard high-valued ones (and the paths leading to them). This has some merit but often the evaluation function will be noisy. A measure is needed which will smoothe out this noise, so the idea of 'penetrance of the tree' has been generalised to 'flow of a node'.

Suppose we have a search tree T . Let $\text{Nod}(x)$ be the number of descendants of node x . (Note that for a leaf $\text{Nod}(x) = 0$). If the path from root r to node s is $x_0, x_1, x_2, \dots, x_n$, where $x_0 = r$ and $x_n = s$, then define $\text{Flo}(s)$ to be

$$\prod_{k=1}^n \frac{\text{Nod}(x_k) + 1}{\text{Nod}(x_{k-1})} \dots\dots (1)$$

Since $\text{Nod}(x_{k-1}) \geq \text{Nod}(x_k) + 1$ ($k=1,2,\dots,n$), we have immediately that $0 < \text{Flo}(x) \leq 1$ for all $x \in T$.

The definition may seem a little arbitrary at first but in fact the measure Flo has the properties we seek. It is high in the 'richer' parts of the tree and low for those nodes which lead only to a few other nodes. Also, as we trace a path from root to leaf the flow is decreasing. Consider two nodes a and b, such that a is on the path leading from the root to b. Then immediately from (1) we have that $\text{Flo}(a) \geq \text{Flo}(b)$.

When examining the values of Flo for the nodes, we are interested in the relative and not the absolute magnitudes of the function. A simple formula gives us an idea of the scale of values; if there are λ leaves on the tree then the average value of Flo for each leaf is λ^{-1} . This follows from the following result:

Lemma If L is the set of leaves of a tree T then

$$\sum_{\ell \in L} \text{Flo}(\ell) = 1$$

Proof Suppose we have two trees T_1 and T_2 for which the lemma is true. Let us form a single tree T by placing a leaf of T_1 , ℓ^* say, and the root of T_2 in coincidence (see Fig 3.3).

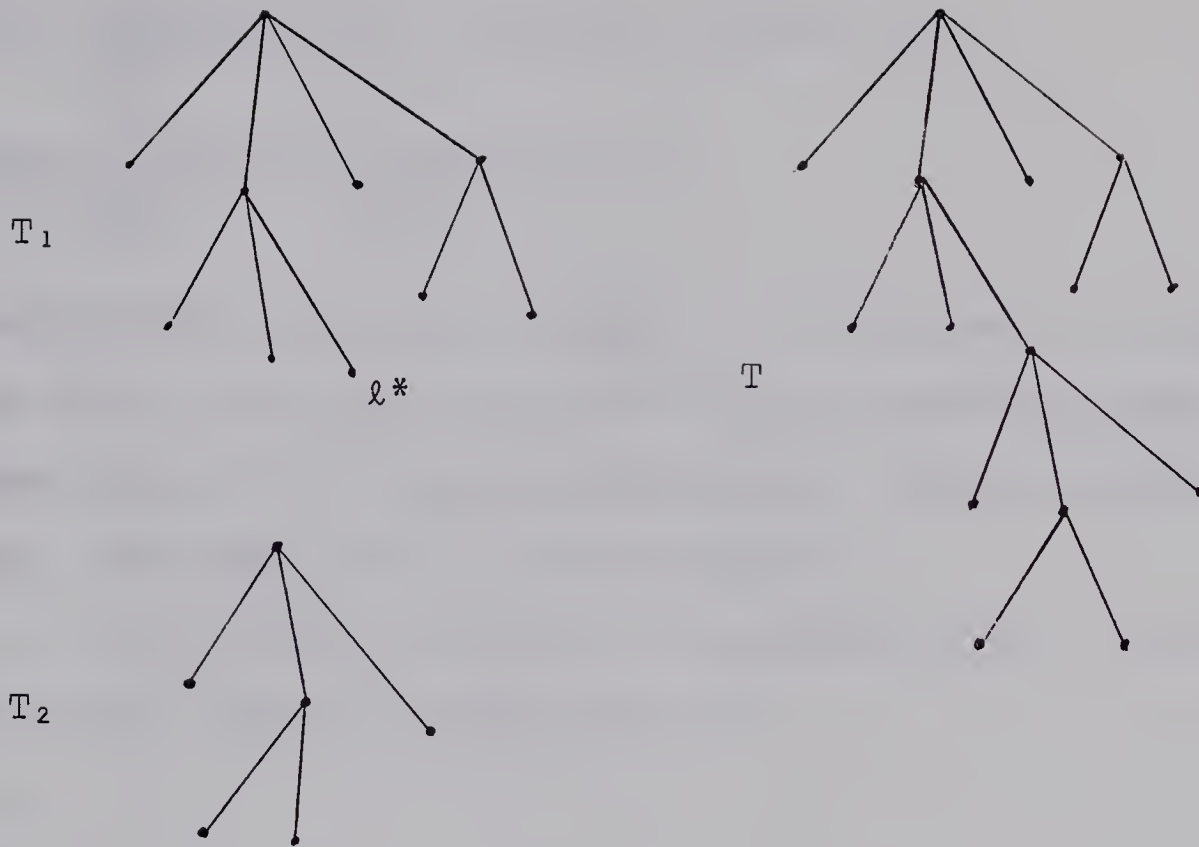


Fig 3.3 Tree T is constructed by joining T_1 to T_2 at leaf l^* .

If L_1 and L_2 are the sets of leaves of T_1 and T_2 , and L is the set of leaves of the tree T then

$$\sum_{l \in L} \text{Flo}(l) = \sum_{j \in L_1 - \{l^*\}} \text{Flo}(j) + \text{Flo}(l^*) \sum_{k \in L_2} \text{Flo}(k)$$

since each term in the second summation on the right hand side must be multiplied by $\text{Flo}(l^*)$ (from (1)).

$$\therefore \sum_{\ell \in L} \text{Flo}(\ell) = 1 - \text{Flo}(\ell^*) + \text{Flo}(\ell^*) = 1$$

$$\text{since } \sum_{j \in L_1} \text{Flo}(j) = \sum_{k \in L_2} \text{Flo}(k) = 1.$$

Now consider any tree of height 1. The lemma is trivial for such a tree, and any tree T can be formed by combining such trees in the way described above. Hence, by induction, the lemma is true for any tree T .

It is worth noting that in general $\text{Flo}(\ell_1) \neq \text{Flo}(\ell_2)$ ($\ell_1 \neq \ell_2$), see for example Fig 3.4.

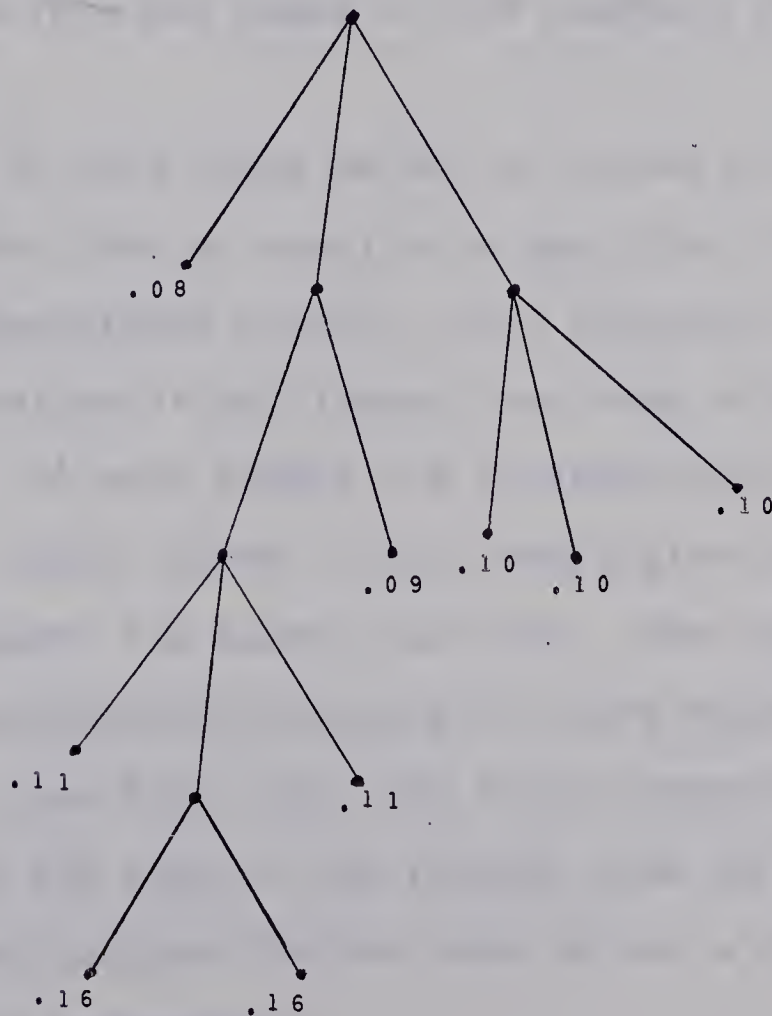


Fig 3.4 A search tree showing the flow value for each leaf.

The Function PRUNE

This is the function which reduces the search tree produced by SEARCH and described by FIGS to a smaller structure - a connected partial subgraph. This reduction is done in three stages:

(i) Trace the path to the most promising node. Take a certain number of steps along this path and display them. Keep the subtree which is rooted at the end of this path and discard the rest.

(ii) Paths are traced from the new root to leaves with values less than or equal to some 'leaf threshold'. These paths form the basis of the subgraph produced by PRUNE.

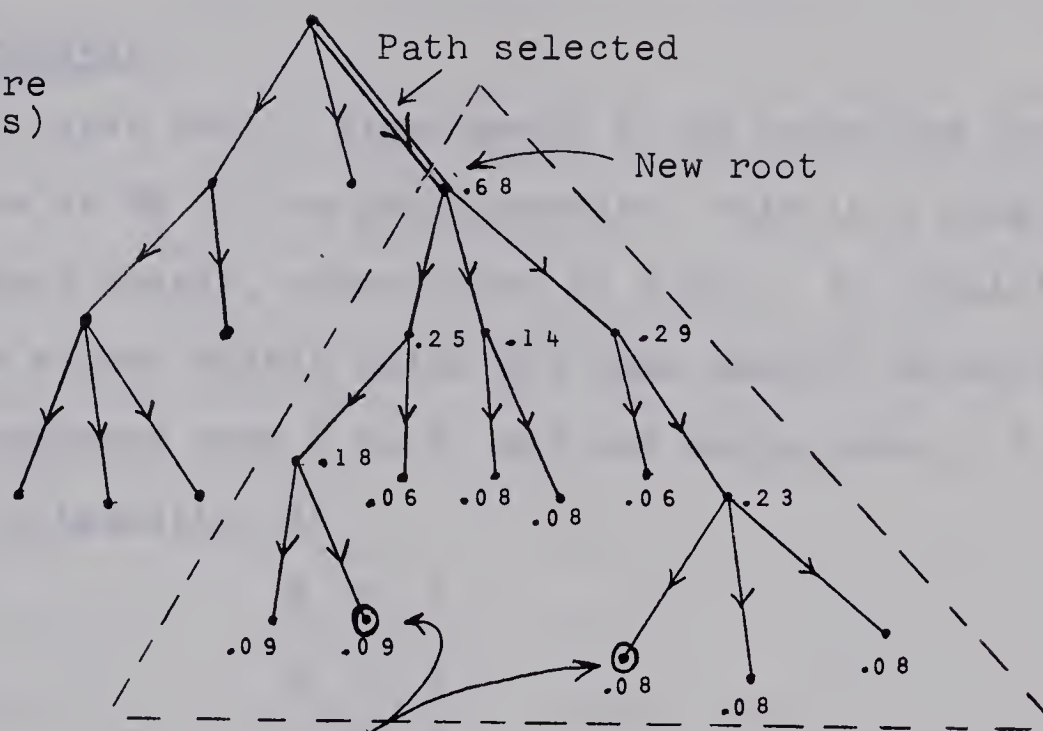
(iii) To this basis we add all nodes with a flow value greater than or equal to a specified 'flow threshold'.

The user first enters a leaf threshold. PRUNE then lists the values of all leaves less than or equal to this threshold. It also prints the intersection of the paths leading to these leaves. This should give some idea of how 'divergent' the search has been. The user then supplies the parameters necessary for tree reduction to take place, viz. new root, leaf and flow thresholds. When this is complete the size of the reduced tree is displayed and the user can replace the old tree or try a different set of reduction parameters.

The value of each undeveloped node is recomputed in

PRUNE so that the user can change the evaluation function during problem-solving if he wishes.

Old Tree:
(numbers are
node flows)



Nodes with value below threshold
specified for leaves.

New Tree:

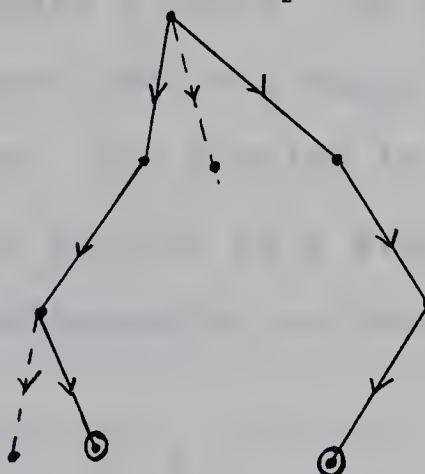


Fig 3.5 PRUNE acting on a search tree. The nodes in the old tree outside the triangle are discarded. In the new tree the paths to the low valued nodes are shown with solid lines. The broken lines lead to nodes with sufficiently high flow values to be retained. (In this example the flow threshold is 0.09).

CHAPTER IV

EXPERIMENTS WITH THE GRAPH TRAVERSER

The Eight-puzzle

The first set of experiments to be described involves application of GT to the Eight-puzzle. This is a simple sliding-block puzzle, often found as a toy. It consists of a large square within which are some smaller movable squares, numbered from 1 to 8, and one empty space. A typical configuration is

8	0	3
2	1	4
5	6	7

where the zero represents a space. We can slide any of the neighbouring squares into the empty space leaving behind a new empty space. The problem is to transform one given arrangement into another by a succession of these moves. The 'goal' configuration we have arbitrarily chosen to be

1	2	3
8	0	4
7	6	5

For any particular goal arrangement, including this one, only half the possible starting arrangements give a soluble problem. (For a combinatorial treatment see Schofield 1967).

It is easy to see how we can make the problem fit the Graph Traverser scheme. Each configuration corresponds to a node of the graph, represented by a 3x3 integer matrix, and each 'move' corresponds to an arc of the graph. A node will have two, three or four immediate descendants depending on whether the empty square is in a corner, in the middle of a side or in the centre of the square.

There are two terms in the evaluation function used. For each piece we find the 'city-block distance' from its goal position. In the example shown above the distance of the piece numbered 2 from its correction location is 2 and piece number 8 is a distance of 1 away from its 'home'. P is the sum of these eight distances. S measures the degree to which the pieces are in the correct sequence. We proceed cyclically around the edge, ignoring the empty space*, and each numbered piece scores 2 if it is not preceded by the required piece. In the configuration above piece number 3 is not preceded by the correct piece (2) whilst the correct piece does precede 4 (3). The evaluation function used is $P + 5S$. For the sample configuration shown above $P=9$, $S=13$ and the immediate descendants generated by all possible moves, are

0	8	3	8	3	0	8	1	3
2	1	4	2	1	4	2	0	4
5	6	7	5	6	7	5	6	7

This evaluation function was found by Doran and Michie

* Score 1 if the empty space is not in the centre.

(1966) to be the best available from the selection they tried, which were all of the form $P + wS$ where P and S are as defined above and w is a variable parameter. The Eight-puzzle experiments described here were intended to test different pruning schemes in order to find optimal control parameters and also to compare the modified search algorithm with the original. The behaviour and possible improvement of the evaluation function is not considered and so the same function is used throughout.

Experiment 4.1

Path retention

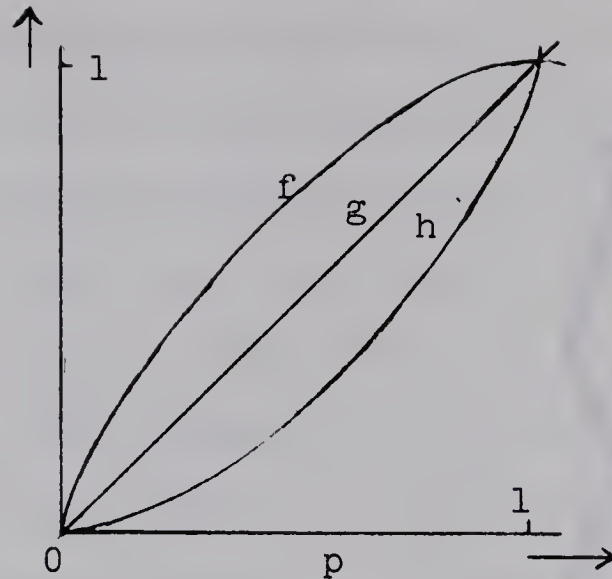
When a partial search is complete and PRUNE is called the first stage of tree reduction is to take a number of steps along the most promising path. How many should we take? In this experiment we compute this number of steps to be taken from the root as a fraction of the total path length from root to most promising node. This fraction is defined as a function of the search penetrance p (see Chapter III) and here we compare the results of using different functions. Three different ones were tried, one was convex, one linear and one a concave function of p . They were:

$$f(p) \equiv \sin \pi p / 2, \quad 0 \leq p \leq 1,$$

$$g(p) \equiv p, \quad 0 \leq p \leq 1,$$

$$h(p) \equiv 2p - \sin \pi p / 2, \quad 0 \leq p \leq 1.$$

Fig 4.1
The retention
functions.



Note that each function is monotonically increasing in the domain shown and takes the values 0 and 1 at the end points.

There is a difficulty however. If the search has ended and there are several nodes with the minimum value, by taking the computed number of steps (using one of the above functions) we may be committing the program to one of these 'most promising' nodes by discarding the path to one or several of the others. The situation is shown in Fig 4.2. To ensure this does not occur we never take any steps past the node from which the paths diverge (depth 2 in the case shown but it may even be the root).

Once the number of steps is decided we must supply PRUNE with the 'leaf' and 'flow' thresholds. For this experiment these are taken to be the value and flow of the most promising node (ties for most promising node are resolved as mentioned in Chapter III, p.32). The maximum tree size was set at 40 nodes to ensure fairly frequent

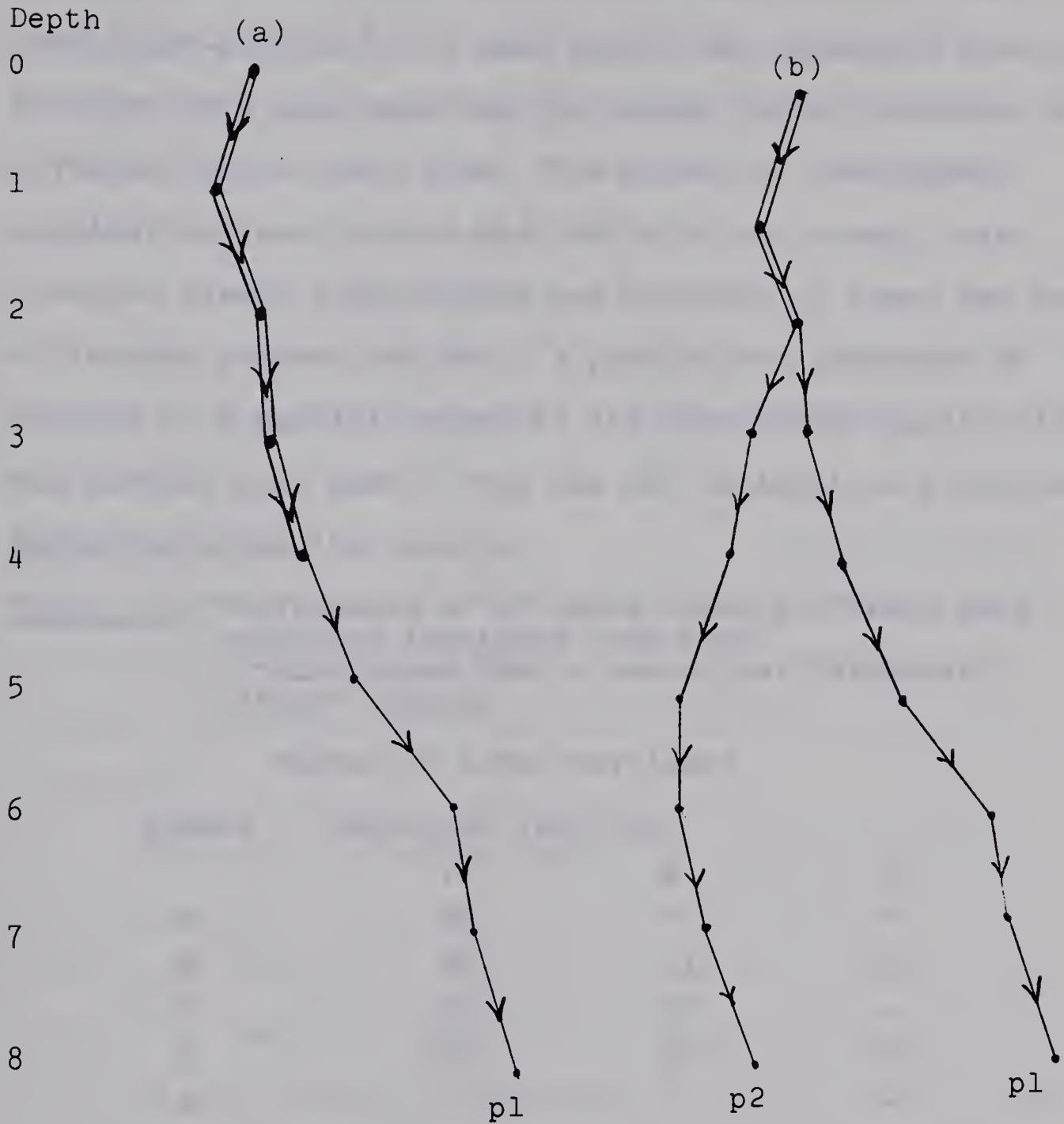


Fig 4.2 Paths traced to most promising nodes after a partial search. In (b) we cannot take more than two steps without committing ourselves to either p1 or p2 and excluding the other from the sub-graph to be constructed.

pruning and GT was applied to 12 randomly selected, solvable Eight-puzzles.* For each puzzle and retention function two runs were made starting the random number generator at different points each time. The number of development applications was counted and the solution attempt which involved fewest applications was recorded if there was any difference between the two. A problem was abandoned at the end of a partial search if 100 development applications had already been made. This was the resignation criterion. Table 4.1 gives the results.

Table 4.1 Performance of GT using three different path retention functions (see text).
'-' indicates that a search was terminated without success.

Number of nodes developed			
Puzzle	Retention function		
	f	g	h
A	82	-	-
B	40	51	43
C	76	57	-
D	60	62	61
E	-	-	-
F	26	26	26
G	18	18	18
H	42	42	42
I	109	109	-
J	89	-	-
K	30	30	30
L	21	21	21
Failures	1	3	5

* These puzzles are listed in Appendix D.

When a puzzle is solved with all three retention functions performance is about the same for each one. This occurs for easy puzzles which require only one or two partial searches. For this set of puzzles 'f' was best and it is interesting to note that

'f' fails implies 'g' fails

and 'g' fails implies 'h' fails, in these particular results. Some further comment follows in Chapter V.

Experiment 4.2

Subgraph detail

Once we have decided what the root of the new tree is to be and have traced paths to it from some of the leaves, which arcs should we add to the basic structure produced?

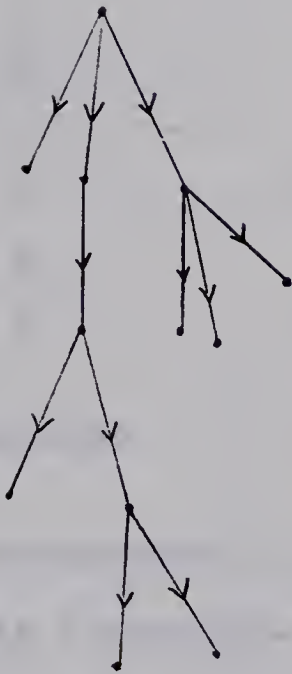
The scheme was explained in Chapter III whereby we add those nodes which have a flow value exceeding some particular threshold. But we still have to choose this threshold. Table 4.2 shows the results of applying GT to the 12 Eight-puzzles of Experiment 4.1 using some different values for the flow threshold, the same evaluation function $P + 5S$ and the retention function 'f'. The experimental details are the same as before. Fig 4.3 illustrates the meaning to be attached to these different flow thresholds.

Fig 4.3 Effect of different flow thresholds on pruning.

(α) threshold = 1.
path to most promising node only



(β) threshold = flow of most promising node.
structure contains some additional arcs



(γ) threshold = 0.
structure contains paths to all leaves descended from selected root

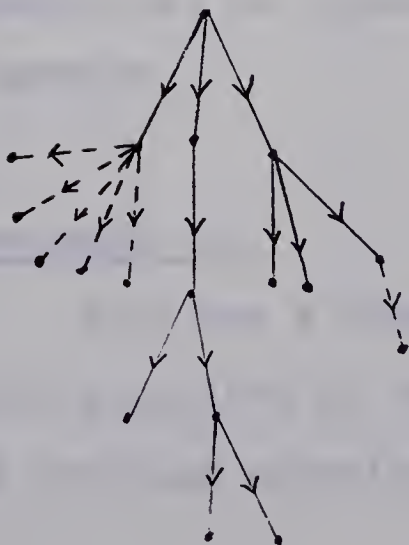


Table 4.2 GT performance with three different flow thresholds.
 α -threshold=0; β -threshold=flow of most promising node; γ -threshold=1.
'-' indicates termination of a search without success.

Number of nodes developed			
Puzzle	Threshold- α	- β	- γ
A	82	82	86
B	40	40	40
C	-	76	103
D	60	60	66
E	-	-	-
F	26	26	26
G	18	18	18
H	42	42	42
I	-	109	-
J	89	89	89
K	30	30	30
L	21	21	21
Failures	3	1	2

Performance is best with threshold- β . If GT succeeds with threshold- α then performance is identical with threshold- β , but threshold- γ may require more applications of DEVELOP.

Experiment 4.3

Search algorithms

This was a comparison between Doran and Michie's search algorithm S1 and the modified form S2 which uses a less thorough matching procedure (the difference between

the two algorithms is explained in Chapter III). Table 4.3 refers to the results of using these two algorithms on two different Eight-puzzles. One problem was 'easy' and the other was 'difficult' (D and E respectively) compared with other Eight-puzzles. Various maximum tree sizes were tried and the best pruning scheme available was used, viz. retention function 'f' in conjunction with flow threshold- β .

Table 4.3 Search time for the solution of two Eight-puzzles using search algorithms S1 and S2 (see text). The time taken by FIGS and PRUNE is not taken into consideration.

Retention function = f

Flow threshold = β

Puzzle	Search algorithm	Tree size	Number of partial searches	CPU Time (secs)
D	S1	40	3	36
D	S2	40	3	18
D	S1	80	2	38
D	S2	80	2	19
D	S1	160	1	38
D	S2	160	1	19
E	S1	100	2	74
E	S2	100	3	40
E	S1	150	2	90
E	S2	150	2	49
E	S1	200	1	79
E	S2	200	1	40

S2 took about half the search time of S1 and only once resulted in an extra partial search, due to keeping

duplicates in the search tree. Note also that search time is approximately constant for each problem and does not depend very much on the number of partial searches taken.

The Queens Problem

The Eight Queens problem is a classic of combinatorial analysis (Netto 1920, Ginsburg 1939). Eight 'queens' are to be placed on an 8x8 chessboard in such a way that no two are mutually attacking. Hence there must be no two of them on a common rank, file or diagonal. There are ${}_{64}C_8$ possible ways of placing the queens on the board i.e. some 4.4 billion, and it can be shown (by a search method to be described later) that 92 of these ways are solutions to the problem. Out of these 92 configurations there are 12 distinct fundamental solutions and from these we can obtain the remainder by reflection and rotation.

Fig 4.4 A solution of nonattacking queens on the 8x8 chessboard.

			X				
					X		
							X
	X						
						X	
X							
		X					
				X			

The problem can be viewed as a simple allocation task with constraints and it was with this new sort of application in mind that the Graph Traverser was used for the problem. It involved a 'property search' as distinct from the 'path search' for solution of the Eight-puzzle. Notice also that the problem is not tied to an 8x8 board but can be posed generally for an NxN board with N 'queens'. GT was used to solve problems with N varying from 4 to 14. The primary purpose was to establish that the Graph Traverser can usefully be applied to such problems and a comparison was made with a known successful technique. This work was completed when the author became aware of the similar (but more thorough) work done by Doran (1968) in connection with the problem of the Travelling Salesman. The author's work adds evidence to support the common approach used since both the problem attacked and the algorithm with which we compare the Graph Traverser are different from those used by Doran.

Firstly we have to convert the Queens problem into a problem graph on which GT can operate. Each possible configuration of the queens on the board maps into a node and is represented by a Boolean matrix. A 1 denotes occupation of a square by a queen and a 0 denotes an empty square. It is clear that we need only consider matrices which have just one non-zero entry in each row and one in each column (i.e. orthogonal matrices). This reduces the

size of the whole problem-graph from 4.4 billion nodes to 20,000 or so for the 8x8 case and we do not lose any solutions this way.

The operator set used on the matrices is the set of row-interchanges involving the first row of these matrices. The function DEVELOP produces a list of all matrices resulting from these interchanges. When applied to a configuration EVALUATE forms a weighted sum of the number of queens on each diagonal; for a diagonal with no queens or one queen score 0, for two queens score 4, for three queens 9, for four queens 16, etc. There will be a score of zero if and only if a goal configuration is located.

Initially a somewhat larger operator set was used, namely the set of all possible row interchanges. For an $N \times N$ board this produced $N(N-1)/2$ operators compared with $(N-1)$ operators when interchanges could only be with the first row. The second operator set is weaker in the sense that it produces fewer connections between the nodes of the graph but it is strong enough to solve the problem. The first set produces so many interconnections on the graph that the time taken to develop a single node was found to be prohibitive.

Experiment 4.4

Queen problem solution, $N = 4$ to 14

Searches were started from randomly selected orthogonal matrices and a time limit of 3 minutes was imposed

on each search. Only one partial search was performed for each starting configuration even if the goal was not located during that one search. Table 4.4 shows the results obtained with the Graph Traverser and also those obtained under the same conditions using backtracking, the search technique mentioned below.

Table 4.4 Solution of the N Queens problem using two different algorithms (see text).

p = estimated probability of success of a single search from a random start
 = (number of successes)/(total searches)

Board size N	Graph Traverser		Backtrack algorithm	
	Mean search time (CPU secs)	p	Mean search time (CPU secs)	p
4	0.9	1	0.3	0.8
5	0.8	1	0.7	1
6	13	1	5	0.6
7	7	1	4	1
8	25	1	16	0.9
9	46	1	34	1
10	58	1	70	0.9
11	147	0.5	95	0.9
12	147	0.5	156	0.3
14*	225	0.2	192	0.2

*Time limit on all searches except N=14 was 3 minutes. For N=14 this was extended to 4 minutes.

Backtracking

Backtracking is a refined method of exhaustive search and is described in detail in Walker 1960 and

Golomb and Baumert 1965. The interested reader should consult these two works for a complete treatment of the method and here we merely discuss application to the problem on hand, viz. the Queens problem.

To simplify matters we consider only a 4×4 board with 4 queens. Suppose we place a queen on the first square of the first rank (Fig 4.5(a)). The first file is now full so we place a second queen on the first available square of the second file. This is square three (Fig 4.5(b)) and now we move to the third file. No queen can be placed anywhere on it without attacking one of the other two so we *back-track* to the second file and move its queen down to the next available square (Fig 4.5(c)). We can now place a

1			

(a)

1		.	
		.	
	2	.	
		.	

(b)

1			
	2		

(c)

1			.
		3	.
			.
	2		.

(d)

1			

(e)

1			
	2		

(f)

		3	
1			
	2		

(g)

		3	
1			
			4
	2		

(h)

Fig 4.5 Solution of the 4×4 Queens problem by backtracking.

queen on the second square of the third file (Fig 4.5(d)) but find that this leaves no available squares on the last file. Neither the third nor the second queen can be moved down their file so we *backtrack* to the first file and advance the first queen (Fig 4.5(e)). We reset the second queen to the first available position on the second file (Fig 4.5(f)) and set the third and fourth queens onto the first available positions of their files (Fig 4.5(g) & (h)). This gives a solution.

The crux of the method is the way we rule out sets of test configurations or candidate solutions. If the first two queens attack each other say, it does not matter where the other two queens are. We cannot possibly have a solution. So if we find an attacking pair for some configuration where only some of the queens have been placed on the board we do not need to investigate this position any further. In this way we build up a configuration component by component and as soon as the constraints on the problem are violated (two queens attack each other) we backtrack and reject all other configurations which might be constructed by adding components to the unsatisfactory one.

A backtracking search was programmed for the N Queens problem in such a way as to start from a random orthogonal configuration. Starting this way, in the midst of the solution space, was found to be more effective than starting at the boundary of the space by setting a queen on

the first square of the first file. Search times were greatly improved. It would appear that the solution space is 'rarefied' around this initial position.

The search times for backtracking and for Graph Traverser solution of the Queens problem varied tremendously. The mean times shown in Table 4.4 are only intended to give the reader an idea of the growth of solution time compared with problem size for the two algorithms. They should not be taken too literally. It had been expected, pessimistically as things turned out, that the backtracking algorithm would be much faster than GT, at least for small board sizes. The Graph Traverser is a general problem solver while the backtracking search program has been written explicitly for the Queens problem. As was mentioned in Chapter I, we usually pay a high price for generality. Surprisingly enough it is not the case here, search times were similar for both programs. It had also seemed that search time for the backtrack algorithm would grow more rapidly with increases in problem size than it would for the Graph Traverser. Nor was this conjecture borne out. A possible explanation is that the number of actual solutions of the problem increases substantially with board size and backtracking profits more from this phenomenon than does the Graph Traverser. For backtracking the rate of discovering solutions is more strongly connected with the density of solutions than with the spatial location of these sol-

utions. For GT it seems to be vice versa.

Missionaries and Cannibals

The Graph Traverser was applied to the well-known problem of missionaries and cannibals (Sholander 1942, Busaacker and Saaty 1965). 'Three missionaries and three cannibals arrive at bank A of a river and must cross to bank B using a rowboat which will accomodate only two people. All of the missionaries and one of the cannibals can row. Is it possible to achieve this transfer through a sequence of crossings such that the cannibals never outnumber the missionaries on either bank unless, of course, the number of missionaries is zero? (The missionaries feel strongly about this rule.)'

The state of the system was represented by a vector giving the number of missionaries and cannibals (rowing and non-rowing type) present on bank A and the location of the boat. The operator set used consisted of all allowable (with regard to who was in the boat) transitions. The evaluation function was extremely simple, 0 if the goal was reached (nobody left on bank A and the boat at bank B) and 1 otherwise. This reduced the GT strategy to systematic search and a test was built in to the SEARCH function so that it would stop if all possible development had been made without location of the goal. The problem was solved and a similar problem involving just one rowing missionary was

proven insoluble. Running times were one or two seconds for each problem.

CHAPTER V

CONCLUSIONS

Pruning Control

In the first two experiments with the Eight-puzzle we were attempting to find optimal control settings for the pruning parameters. The experimental design was to split pruning into two serial tasks;

(i) choose a basic structure for the new subgraph,
and

(ii) add some arcs onto this structure.

We then set about finding optimal settings for each task separately. This is an example of planning as described in Chapter I. Instead of trying all possible control settings for the whole process we set up serial subgoals and solve them individually.

The results showed that over the range of puzzles chosen $f = \sin \pi p / 2$ was the best retention function and threshold- β was the best flow threshold to choose. Retention functions $g = p$ and $h = 2p - \sin \pi p / 2$ did not place enough confidence in the results of the search, and too little information was displayed as output from it while too much was retained in the tree. The new root was too close to the old one so the displayed path was short. Looked at another way, too much information was stored in temporary memory, i.e. the tree, and not enough transferred to more

permanent storage i.e. printed output.

Threshold- α also retained too much information in the tree. The effect of doing this is to allow each search to add only a few nodes to the search tree with which it begins. Problem solving steps are too small and the system can be trapped in a local minimum of the solution space. A promising node is developed and the descendants turn out to have unexpectedly high values. This unsuccessful exploration is cut from the tree at pruning but performed again on the next partial search. During this partial search the tree reaches maximum size before another node in a different area can be developed. In this manner the program is trapped.

The situation with threshold- γ is simpler to understand. This time too little information is retained in the tree and previous work sometimes has to be duplicated.

Heuristic Information Handling Theorems

It is important to note that the results quoted in this work refer only to the limited evidence from the Eight-puzzle. It had been hoped to extend application to the Fifteen-puzzle (a 4×4 version of the Eight-puzzle) and to other problems but the time taken for some preliminary trials was prohibitive. So we have pruning evidence from only one environment. The results are quite good but could they be better? If so, how much better? How would the system per-

form in other environments? These are difficult questions. We need efficiency and information capacity theorems for heuristic systems. At present there is no theoretical way for comparing two such systems. To quote from Holland (1966), "...we are at much the same stage as steam engine designers before the advent of Carnot".

Improvement of the Evaluation Function

As mentioned earlier an attempt was made to find a general method of improving the evaluation function. The approach used was to write the evaluation function as a weighted sum of separate functions each of which evaluated a different feature of the problem-state (cf. the evaluation polynomial in Samuel 1959). A method was sought which would give the best possible setting to these weights. Firstly we remark that there does indeed exist a perfect evaluation function, viz. $d(n, G)$ the distance in the graph from the node n representing the problem-state to the goal node G , assuming of course that the problem is soluble. Finding this actual distance function is impossible in most cases due to the size of the problem graph. Besides, we would have to construct a path from node to goal and the problem would already be solved. The Graph Traverser method depends on finding a function $d^*(n)$ whose value for any node n can be computed by considering just the features of the node and whose value is a good predictor of graph distance

from the node to the goal. To be a little more precise, we would like d^* to satisfy the following condition: for any pair of nodes n_1 and n_2 , $d^*(n_1) > d^*(n_2)$ if and only if $d(n_1, G) > d(n_2, G)$. We try to find a linear combination of the feature functions which will have this property for most pairs of nodes.

Various approaches for improvement of a given evaluation function were tried, mostly on paper. The general idea was to perform a partial search and adjust the weights, depending on the results of that search, before proceeding to the next search. The adjustments were based on different forms of correlation between the values of the terms in the evaluation function and the distances from the root of the nodes on the most promising path found. This followed along the lines suggested by Doran (1967). Some other features of the graph as well as distance were involved in some of the schemes. No success was attained, the schemes were too unwieldy to use or produced meaningless or inconsistent results. It seems that we need to gather data from more than a single search before weight adjustment is attempted. The difficulty appears to be that these 'search-adjust' schemes are unstable. The standard hill-climbing approach, also outlined by Doran (op.cit.), is more promising. Sequences of partial searches are carried out with different weight settings and the penetrance, or some other figure of merit for each search, is used to converge onto

an optimal setting. This is a lengthy procedure but the small amount of work considered using the other approach indicates that it may well be necessary in order to provide convergence. We are still left to pick the different feature functions anyway and this is not particularly easy. Once this is done hill-climbing may not be the complete answer. I think a remark of Minsky's is relevant here, "Certainly, in our own intellectual behavior we rarely solve a tricky problem by a steady climb towards success. I doubt that in any one simple mechanism, e.g. hill-climbing, will we find the means to build an efficient and general problem-solving machine. Probably, an intelligent machine will require a variety of different mechanisms. These will be arranged in hierarchies, and in even more complex, perhaps recursive, structures. And perhaps what amounts to straightforward hill-climbing on one level may sometimes appear (on a lower level) as the sudden jumps of 'insight'." (Minsky 1961)

Certainly an interesting problem for Graph Traverser 2 would be improving the evaluation function of Graph Traverser 1. To do so we would need an evaluation function for Graph Traverser 2 and Graph Traverser 3 might try to improve that, and so on, ad infinitum.

A Modified Strategy for Property-Search Problems.

GT has been successful in solving the Queens problem using the very simple approach described in Chapter IV. The penetrances of the successful searches were almost always high (0.8 - 1.0) and those of the unsuccessful ones were rarely less than 0.4 say. It seems quite likely that even the unsuccessful searches were on their way to finding a solution (though not quite as directly as the successful ones) and were not wandering circuitously around the problem graph.

As remarked before, the problem involved a property search, yet we are finding a path from start to goal. Is this really necessary? We might do well just to keep a list of the leaves of the tree and discard a node as soon as it was developed. If we do this there is a chance that we will loop around by developing a leaf, producing one of its ancestors over again and then repeating our previous developments. For many problems this will happen but if the evaluation function is good, as it seems to be for the Queens problem, it should not occur too often. We expect to encounter a series of nodes with decreasing values and therefore should not loop. A simple and fast test could be introduced to check for cycling. This would replace the earlier matching, which is still expensive despite the 50% gain.

Another approach would be to discard nodes with

high values (higher than that of the current most promising node say) as they were developed. This leads us into the concepts of operator selection and partial development. These ideas are treated in Michie 1967 and Doran 1968 and so we refer the reader to these works. We make just one point about operator selection; despite the author's intention of not becoming involved in the topic it was found necessary to apply an elementary form to the Queens problem operators by selecting $(N-1)$ of them instead of $N(N-1)/2$ (see p.53). This reduced search times enormously. The technique of operator selection is a powerful and necessary one it seems, and from Doran's promising results with the Travelling Salesman problem (Doran 1968) it seems that the same can be said for partial development of nodes.

Role of the Graph Traverser

Introducing the Graph Traverser in an interactive form does not change its nature in any fundamental way but it has produced some new ideas. A large amount of work has been done on setting up a good control scheme with particular emphasis on pruning. The user now plays an important role in deciding which results are useful and which are not after each partial search.

The major role of the system is probably that of a testing ground for new ideas in problem-solving, learning, machine intelligence etc. Hence its construction is as

flexible as possible.

In Chapter I we stressed the importance of problem representation. For GT we determine the structure of the problem graph by the state representation which we choose and the function DEVELOP that we write. This is the environment in which GT sets to work, guided by the function EVALUATE. Sometimes the same problem can be represented by different environments and some may be more amenable to problem solution than others. So far all the efforts to improve the Graph Traverser have been concerned with the strategy it adopts. These efforts are necessary but a rich and unexplored field for future work must surely be the investigation of different problem representations. The Graph Traverser could provide a common strategy and operational system for this work, a role of no mean importance.

REFERENCES

- Amarel, S. (1966). On the mechanisation of creative processes. IEEE Spectrum (April 1966).
- Amarel, S. (1968). On the representation of problems and goal-directed procedures for computers. Working paper (Computer Theory Research Group, RCA Laboratories, Princeton, New Jersey). To be published in Proceedings of the First Annual Symposium of the Society for Cybernetics (forthcoming).
- Berge, C. (1962). The theory of graphs and its applications. Alison Doig, Trs. Methuen, London.
- Bobrow, D.G. (1964). A question-answering system for high school algebra word problems. AFIPS Conference proceedings - 1964 Fall Joint Computer Conference I, 591-614. Spartan Books, Baltimore.
- Busaacker, G.B. & Saaty, T.L. (1965). Finite Graphs and Networks. McGraw-Hill, New York.
- Doran, J.E. & Michie, D. (1966). Experiments with the Graph Traverser program. Proc. Royal Soc. (A) 294, 235-259.
- Doran, J.E. (1967). An approach to automatic problem-solving. Machine Intelligence 1, 105-123. Collins, N.L. & Michie, D. (eds). Oliver and Boyd, Edinburgh.

- Doran, J.E. (1968). New developments of the Graph Traverser. Machine Intelligence 2, 119-135. Dale, E. & Michie, D. (eds.). Oliver and Boyd, Edinburgh.
- Evans, T.G. (1964). A Heuristic Program to Solve Geometry-Analogy Problems. Proc. Spring Joint Comp. Conf. 25, 1964, 327-338.
- Falkoff, A.D. & Iverson, K.E. (1968). APL\360 User's Manual. IBM Corp., Yorktown Heights, New York.
- Feigenbaum, E. & Feldman, J. (eds.). (1963). Computers and Thought. McGraw-Hill, New York.
- Floyd, R.W. (1967). Non-deterministic algorithms. J. Ass. Comp. Mach. 14 (October 1967).
- Gelernter, H. & Rochester, N. (1958). Intelligent behaviour in problem-solving machines. IBM J. Res. Dev. 2 (4), 336-345.
- Gelernter, H. (1959). Realisation of a geometry theorem-proving machine. Proceedings of the International Conference on Information Processing (ICIP), UNESCO, Paris. (Reprinted in Feigenbaum & Feldman, above).
- Ginsburg, J. (1939). Gauss's arithmetisation of the problem of 8 queens. Scripta Math. 5, 63-66.

- Golomb, S.W. & Baumert, L.D. (1965). Backtrack programming. J.Ass. Comp. Mach. 12, 516-524.
- Green, B.G., Wolf, A.K., Chomsky, C. & Laughery, K. (1961). Baseball: An Automatic Question-Answerer. Proc. Western Joint Computer Conference, 19, 1961.
(Reprinted in Feigenbaum & Feldman, above.)
- Holland, J.H. (1965). Some practical aspects of adaptive systems theory, in Electronic Information Handling Kent, A. & Taulbeen, O. (eds.). Spartan Books, Washington, D.C.
- Iverson, K.E. (1962). A Programming Language. John Wiley & Sons, Inc., New York.
- Kirsch, R.A. (1964). Computer Interpretation of English Language Text and Picture Patterns. IEEE Trans. Elec. Comp. EC-13, 363-376.
- Kuck, D.J. & Krulee, G.K. (1964). A Problem-Solver with Descriptive Inputs, in Computer and Information Sciences. Tou, J.T. & Wilcox, R.H. (eds.). Spartan Books, Washington, D.C.
- Lindsay, R.K. (1963). Inferential Memory as the Basis of Machines Which Understand Natural Language, in Computers and Thought, Feigenbaum & Feldman (eds.), above.

- Michie, D. (1966). Game-playing and game-learning automata, in Advances in Programming and Non-Numerical Computation. Fox, L. (ed.). Pergamon Press, London.
- Michie, D. (1967). Strategy-building with the Graph Traverser. Machine Intelligence 1, 135-152. Collins, N.L. & Michie, D. (eds.). Oliver & Boyd, Edinburgh.
- Minsky, M.L. (1961). Steps toward artificial intelligence. Proc. IRE, 1961. 49, 8-30. (Reprinted in Feigenbaum & Feldman, above).
- Netto, E. (1920). Lehrbuch der Kombinatorik. Chelsea Publishing Company, New York.
- Newell, A. & Ernst, G. (1965). The search for generality. Proc. IFIP Congress 65. 1, 17-24. Spartan Books, New York.
- Newell, A. & Simon, H.A. (1956). The Logic Theory Machine. IRE Transactions on Information Theory IT-2 (3), 61-79.
- Newell, A. & Shaw, J.C. (1957). Programming the Logic Theory Machine. Proc. Western Joint Comp. Conf. 218-239.

- Newell, A., Shaw, J.C. & Simon, H.A. (1958). Chess-playing programs and the problem of complexity. IBM J. Res. Dev. 2 (4), 320-335. (Reprinted in Feigenbaum & Feldman, above).
- Newell, A., Shaw, J.C. & Simon, H.A. (1959). Report on a general problem-solving program. Proceedings of the International Conference on Information Processing (ICIP), UNESCO, Paris.
- Newell, A., Shaw, J.C. & Simon, H.A. (1960). A variety of intelligent learning in a general problem-solver, in Self-Organising Systems. 153-189. Yovitts, M. & Cameron, S. (eds.). Pergamon Press, New York.
- Nilsson, N.J. A new method for searching problem-solving and game-playing trees. Working paper (Artificial Intelligence Group, Stanford Research Institute). IFIP 68 Congress, Edinburgh.
- Raphael, B. (1964). A computer program which 'understands'. AFIPS Conference Proceedings - 1964 Fall Joint Computer Conference. 1, 577-589. Spartan Books, Baltimore.
- Samuel, A.L. (1959). Some studies in machine learning using the game of checkers. IBM J. Res. Dev. 3, 211-229. (Reprinted in Feigenbaum & Feldman, above).

- Schofield, P.D.A. (1967). Complete solution of the eight-puzzle. Machine Intelligence 1. 125-133. Collins, N.L. & Michie, D. (eds.). Oliver & Boyd, Edinburgh.
- Shannon, C.E. (1950). Programming a digital computer for playing chess. Phil. Mag. 41, 356-375.
- Sholander, M.C. (1942). The Linear Graph. Amer. Math. Monthly 43, 543-545.
- Simon, H.A. (1963). Experiments with a heuristic compiler. J. Ass. Comp. Mach. 10, 4, 493-506.
- Slagle, J.R. (1963). A heuristic program that solves symbolic integration problems in freshman calculus. J. Ass. Comp. Mach. 10, 4, 507-520. (Reprinted in Feigenbaum & Feldman, above).
- Slagle, J.R. & Bursky, P. (1968). Experiments with a multi-purpose theorem-proving heuristic program. J. Ass. Comp. Mach. 15, 1, 85-99.
- Turing, A.M. (1953). pp. 288-295 of Faster than Thought. B.V. Bowden (ed.). Pitman, London.
- Walker, R.L. (1960). An enumerative technique for a class of combinatorial problems. Amer. Math. Soc. Proc. Symp. Appl. Math. 10, 91-94.

APPENDIX A

Graph-Theoretic Terms

A *graph* is a set of *nodes*, any two of which may or may not be connected by an *arc*. An arc may be *directed* (i.e. point from one node to another) or it may have no associated direction. If all its arcs are undirected a graph is said to be *symmetric*. A *path* from one node to another is a sequence of arcs joining the first node to the second. Any directed arcs in the sequence must point 'forwards'. A graph is *connected* if every pair of distinct nodes is joined by some sequence of arcs (called a *chain*), the direction of these arcs being ignored. The *length* of a path is the number of arcs contained in it. The distance between two nodes is the length of the shortest path between them (if a path exists). A *tree* is a connected graph with no 'circuits' or 'loops'. These terms are illustrated in Figs A.1 and A.2.

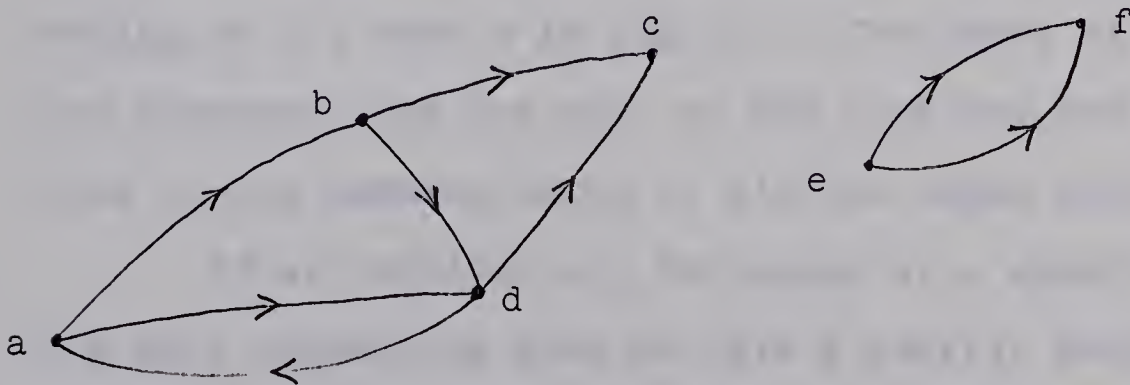


Fig A.1 A graph G with nodes $\{a, b, \dots, f\}$ and 8 directed arcs. G is not connected since there is no path from a to e for example.

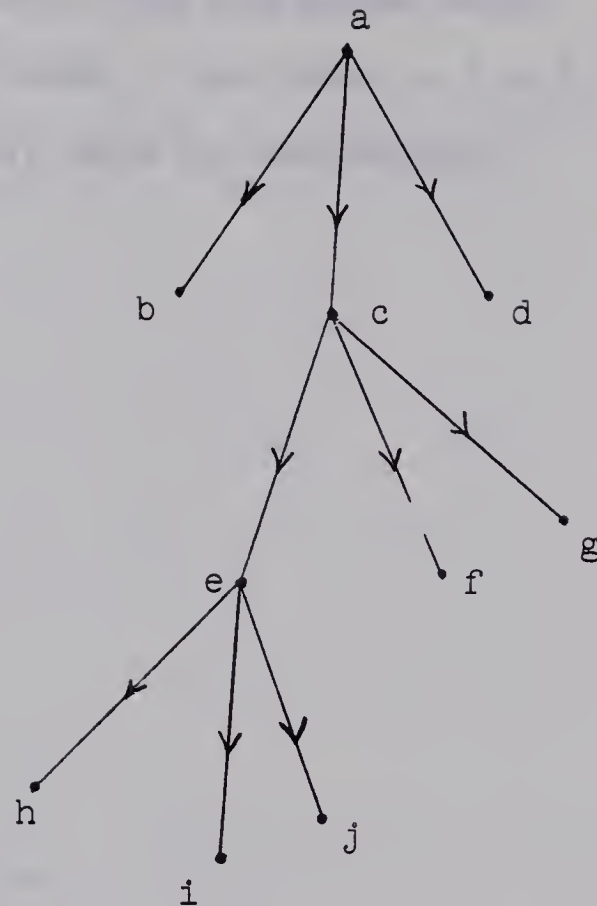


Fig A.2 A tree T. The height is 3.

A *leaf* is a node which is the extremity of a single arc only, node b for example in Fig A.2. The *root* of a tree is that node of the tree which has no directed arcs ending on it, node a in Fig A.2. The *depth* of a node is its distance from the root of the tree and the *height* of a tree is the maximum depth of all the nodes within the tree.

If we consider all the nodes of a graph and some of the arcs connecting them we have a *partial graph* of the original. If we consider some of the nodes and all the arcs connecting those nodes then we have a *subgraph*. The *descendants* of a node are all those nodes which can be

reached by a path from the given node. In Fig A.2 the descendants of node c are nodes e,f,g,h,i and j. Nodes b and d (leaves) have no descendants.

APPENDIX B

Implementation of GT in a Time-Sharing System.

The algorithm has been programmed in APL\360. The APL language was first defined by Iverson in 'A Programming Language' (1962) and has since been developed in collaboration with Falkoff and others. Iverson's language is a notation for the formal description of algorithms or programs. Of necessity it has a considerable syntactic structure of its own and for this reason it is called a 'programming language'.

APL has been implemented as a computer programming language for the IBM System 360 under the name APL\360. It is used here on an IBM 360/67 with remote typewriter terminals. When a user signs on he has immediate access to an active workspace of about 8,000 words. He can also communicate with several 'inactive' workspaces of the same size (his own private library). From these he can copy previously defined programs and data into the active workspace and he can store work from the active workspace into his library for later retrieval.

System commands have as their objects the structures which comprise the system and control functions and information relating to the state of the system. APL operations on the other hand deal with transformation of abstract objects such as numbers and symbols whose practical sig-

nificance depends upon the interpretation placed on them. These APL operators are unambiguously defined and powerful in range. Using them one can build up simple statements and these in turn may be incorporated into defined functions.

The system operates in real-time conversational mode. This feature, coupled with the powerful spectrum of operators and simplicity of function definition and editing procedures, makes for ease of algorithm modification and testing. But the system uses an interpreter so running times are high. The small workspace size has sometimes been a handicap as well as the lack of list-processing facilities. These disadvantages are easily outweighed (for the work described in this thesis at least) by the flexibility of the system. Using conventional batch-processing, with at most two runs a day, it is doubtful if half the work described here could have been completed in double the available time.

APPENDIX C

Solution of an Eight-puzzle and Program Listings.

```

)COPY GRAPH GT                load GT module
SAVED  18.06.37 17/04/69
)COPY GRAPH EIGHTΔPUZZLE      load Eight-puzzle module
SAVED  18.06.37 17/04/69
SETLINK 44                    set random number generator
16807
PUZZLEΔL←(3 3)ρ8 7 2 4 3 0 6 1 5
SET                            initialise GT
ENTER START STATE
□:
    PUZZLEΔL
F=3  3                        state-matrix size
SYSL=35                       upper bound on tree size
MAXJ=30                       lower bound on tree size
MAXT=60                       maximum search time (secs)
DISPP=1                       path display parameter

    SEARCH
SEARCH 1
*****
SPACE LIMIT
GOAL= 0
SEARCH TIME= 4.8 SECS.

    FIGS
ROOT VALUE= 88
MOST PROMISING= 50
AV. UNDEVELOPED VALUE= 67.36
PATH LENGTH= 16
-----
TREE:
    DEVELOPED NODES 16/30
    UNDEVELOPED    14/30
        HEIGHT      16
        PENETRANCE   1
    DEV APPLICATIONS 16
FLO(M.P.NODE)= 0.288

```


PRUNE
ENTER LEAF THRESHOLD
□: 55
VALUE OF LEAF
53
50
COMMON PATH= 1 4 6 9 10 11 13 14 18 nodes in
DIFFERENT THRESHOLD? common
NO portion of
ENTER ROOT the two
□: paths
13
ENTER LEAF THRESHOLD
□: 55
ENTER FLOW THRESHOLD
□: 0.2
NEW TREE SIZE IS 12
REPLACE TREE?
YES
RETAINED PATH LENGTH= 6

8 7 2
4 3 0
6 1 5

8 7 2
4 0 3
6 1 5

8 7 2
4 1 3
6 0 5

nodes in retained path

8 7 2
4 1 3
0 6 5

8 7 2
0 1 3
4 6 5

0 7 2
8 1 3
4 6 5

SEARCH

SEARCH 2

GOAL FOUND

1	2	3
8	0	4
7	6	5

GOAL= 1

SEARCH TIME= 2.2 SECS.

FIGS

ROOT VALUE= 64

MOST PROMISING= 0

AV. UNDEVELOPED VALUE= 37.44

PATH LENGTH= 15

TREE:

DEVELOPED NODES 15/24

UNDEVELOPED 9/24

HEIGHT 15

PENETRANCE 1

DEV APPLICATIONS 6

FLO(M.P.NODE)= 0.225

DAPV= 16 6

development applications
per search
time per search

+ /DAPV= 22

TIMEV= 4.8 2.2

+ /TIMEV= 6.9

P←ROOT PATH L

path to goal

		P													
1	2	3	5	6	7	8	9	10	11	12	16	18	20		
		22		24											

▽DEVELOP[□]▽

▽ S08←DEVELOP P08;MV;POSM;ZP;K08;N08

[1] MV←(∧/POSMε1SQ)/[1]POSM+GRID+(4 2)ρZP+1+(SQ,SQ)τ(1+(,P08)10)

[2] S08←((N08+(ρMV)[K08+1]),SQ,SQ)ρP08←(SQ,SQ)ρP08

[3] S08[K08;ZP[1];ZP[2]]←P08[MV[K08;1];MV[K08;2]]

[4] S08[K08;MV[K08;1];MV[K08;2]]←0

[5] →(2+I26)×1N08≥K08←K08+1

▽

T[P[18];;]			nodes in path to goal		
7	0	2			
8	1	3			
4	6	5			
			T[P[8+18];;]		
7	1	2	1	2	3
8	0	3	7	8	5
4	6	5	4	0	6
7	1	2	1	2	3
8	0	3	7	8	5
4	6	5	0	4	6
0	1	2	1	2	3
7	8	3	0	8	5
4	6	5	7	4	6
1	0	2	1	2	3
7	8	3	8	0	5
4	6	5	7	4	6
1	2	0	1	2	3
7	8	3	8	4	5
4	6	5	7	0	6
1	2	3	1	2	3
7	8	0	8	4	5
4	6	5	7	6	0
1	2	3	1	2	3
7	8	5	8	4	0
4	6	0	7	6	5
			1	2	3
			8	0	4
			7	6	5

```

    ▽EVALUATE[ ] ▽
    ▽ V←EVALUATE M;P;L;U;S
    [1] P←+ / + / |HOME[L/M;]-(L←0≠M←,M)/[1]SLOT
    [2] S←(0≠M[5])+2×+ / ~1=8 | (1ϕU)-U←(0≠U)/U←M[1 2
        3 6 9 8 7 4]
    [3] V←P,W8×S
    ▽
```


Variables used in EVALUATE (for Eight-puzzles)

HOME		SLOT		GRID	
1	1	1	1	-1	0
1	2	1	2	1	0
1	3	1	3	0	1
2	3	2	1	0	-1
3	3	2	2		
3	2	2	3		SQ
3	1	3	1	3	
2	1	3	2		F
		3	3	3	3

```

      VSEARCH[ ]V
  V SEARCH;TIMEON;N;ADDN;DESN;JNEW;L1;L2
[1]  ('SEARCH ' ; (~GOAL←DAP←0)+ρTIMEV)
[2]  8ρ '*'
[3]  TIMEON←I21
[4]  ROOT←(0=J↑PAR)/ιJ
[5]  SEA1:→SEA2×ι1≥ρL←((J↑VAL)=MINV←ι/J↑VAL)/ιJ
[6]  L←L[?ρL←(DEP[L]=ι/DEP[L])/L]
[7]  SEA2:→SEA3×ιMINV=0
[8]  →SEA4×ιMINV≥M1
[9]  →SEA5×ιMAXJ≤J
[10] →SEA6×ιMAXT≤((I21)-TIMEON)÷60
[11] →SEA7×ι~((J↑VAL)≥M1)MATCHES T[L;;]
[12] →SEA1,VAL[L]←M3
[13] SEA7:DESN←DEVELOP T[L;;]
[14] DAP←DAP+1
[15] L1←(ρDESN)[1]ρROOT=L
[16] L2←~^/^/DESN=(ρDESN)ρT[1[PAR[L];;]
[17] ADDN←L1∨L2
[18] →SEA8×ι0<N←+/ADDN
[19] →SEA1,VAL[L]←M2
[20] SEA8:JNEW←J+N
[21] DEP[J+ιN]←Nρ1+DEP[L]
[22] PAR[J+ιN]←NρL
[23] T[J+ιN;;]←ADDN/[1]DESN
[24] SEA9:VAL[J+1]←+/EVALUATE T[J+1;;]
[25] →SEA9×ιJNEW>J+J+1
[26] →SEA1,VAL[L]←M1
[27] SEA3:('GOAL FOUND ' ;T[L;;])
[28] →SEA10,GOAL←1
[29] SEA4:→SEA10,ρ□←'EXHAUSTIVE SEARCH COMPLETED.....NO
      SUCCESS'
[30] SEA5:→SEA10,ρ□←'SPACE LIMIT'
[31] SEA6:'TIME LIMIT'
[32] SEA10:('GOAL= ' ;GOAL)
[33] TIMEV←TIMEV,((I21)-TIMEON)÷60
[34] ('SEARCH TIME= ' ;1 RND TIMEV[ρTIMEV];' SECS.')
[35] DAPV←DAPV,DAP

```



```

      ∇MATCHES[□]∇
    ∇ R←L MATCHES M;E;P
  [1] M←FρM
  [2] E←2ρ1
  [3] P←ιJ
  [4] →(Φ1,(^/(1 1)=E←1+F↑1+F↓E-1),(~v/L←T[P←L/P;E[1];E[
    2]])=M[E[1];E[2]]))/(5 6 4)
  [5] →R←0
  [6] R←1

```

∇

```

      ∇FLOW[□]∇
    ∇ FLO←FLOW;DEPTH;H;LEAF;NODESL;NODESH;DES;NOD
  [1] DEPTH←H←[ /J↑DEP
  [2] NOD←FLO←Jρ0
  [3] NODESH←LEAF←(H=J↑DEP)/ιJ
  [4] NOD[LEAF]←(ρLEAF)ρ1
  [5] FLOW1:NODESL←NODESH
  [6] NODESH←((DEPTH-1)=J↑DEP)/ιJ
  [7] NOD[NODESH]←1+(NODESH◦.=PAR[NODESL])+.×NOD[NODESL]
  [8] FLO[NODESL]←NOD[NODESL]÷NOD[PAR[NODESL]]-1
  [9] →(0<DEPTH←DEPTH-1)/FLOW1
  [10] FLO[ROOT]←1
  [11] DEPTH←2
  [12] FLOW2:DES←(DEPTH=J↑DEP)/ιJ
  [13] FLO[DES]←FLO[DES]×FLO[PAR[DES]]
  [14] →(H≥DEPTH←DEPTH+1)/FLOW2

```

∇

```

      ∇IF[□]∇
    ∇ R←X IF Y
  [1] R←Y/X

```

∇

```

      ∇RND[□]∇
    ∇ R←N RND X
  [1] R←(10*-N)×[0.5+X×10*N]

```

∇

```

      M1
    77777
      M2
    88888
      M3
    99999

```

development-status variables
(from SEARCH)


```

    VPRUNE[ ] V
  V PRUNE; LEAFTH; L; COMM; P; R; RT; NLST; FLOTH; NDEP; NPAR;
    NVAL; NT; PL; LL
[1] PRUNE9: ('ENTER LEAF THRESHOLD')
[2] I←1
[3] LEAFTH←□
[4] L←((LEAFTH≥J↑VAL)^(~(1J)∈PAR))/1J
[5] →(0=ρL)/PRUNE9
[6] 'VALUE OF LEAF'
[7] COMM←ROOT PATH L[1]
[8] PRUNE1:(VAL[L[I]])
[9] P←ROOT PATH L[I]
[10] COMM←((1ρCOMM)≤+/COMM∈P)/COMM
[11] →((ρL)≥I←I+1)/PRUNE1
[12] ('COMMON PATH= '; COMM)
[13] 'DIFFERENT THRESHOLD?'
[14] →('N'≠(R←,□)[1])/1
[15] PRUNE2:'ENTER ROOT'
[16] RT←□
[17] 'ENTER LEAF THRESHOLD'
[18] LEAFTH←□
[19] L←(LEAFTH≥J↑VAL)/1J
[20] NLST←RT PATH L[I+1]
[21] PRUNE3:→((ρL)<I←I+1)/PRUNE4
[22] →PRUNE3,NLST←NLST,(~P∈NLST)/P←RT PATH L[I]
[23] PRUNE4:'ENTER FLOW THRESHOLD'
[24] FLOTH←□
[25] I←1
[26] PRUNE5:NLST←NLST,((NLST[I]=J↑PAR)^(~(1J)∈NLST)^(
    FLOTH≤J↑FLO)^(M2≠J↑VAL)^(M3≠J↑VAL))/1J
[27] →((ρNLST)≥I←I+1)/PRUNE5
[28] NDEP←DEP[NLST]-DEP[RT]
[29] NPAR←((1+ρNLST)|NLST1PAR[NLST])
[30] NVAL←(ρNLST)ρ0
[31] I←1
[32] PRUNE6:→PRUNE7×1(M1=VAL[NLST[I]])^(^(NLST[I]=J↑
    PAR)/1J)∈NLST)
[33] NVAL[I]←+/EVALUATE T[NLST[I];;]
[34] →PRUNE8
[35] PRUNE7:NVAL[I]←M1
[36] PRUNE8:→((ρNLST)≥I←I+1)/PRUNE6
[37] ('NEW TREE SIZE IS '; ρNLST)
[38] →PRUNE2×1'Y'≠(R←□,□←'REPLACE TREE?')[1]
[39] ('RETAINED PATH LENGTH= '; PL←1+ρLL←ROOT PATH RT)
[40] LL←1+LL
[41] DISPP/T[LL;;]
[42] VAL←DEP←PAR←FLO←10
[43] NT←T[NLST;;]
[44] T[1ρNLST;;]←NT
[45] NT←10
[46] J←ρNLST
[47] VAL←NVAL,(SYSL-J)ρ0
[48] DEP←NDEP,(SYSL-J)ρ0
[49] PAR←NPAR,(SYSL-J)ρ0
[50] ROOT←(0=J↑PAR)/1J

```



```

      ▽SET[ ] ▽
    ▽ SET
[1]   DAPV←TIMEV←10
[2]   VAL←DEP←PAR←SYSLρ0
[3]   'ENTER START STATE'
[4]   T←(SYSL,F)ρ0
[5]   VAL[1]←+/EVALUATE T[J+1;;]←Fρ□
[6]   ('F=';F)
[7]   ('SYSL=';SYSL)
[8]   ('MAXJ=';MAXJ)
[9]   ('MAXT=';MAXT)
[10]  ('DISPP=';DISPP)
    ▽

      ▽FIGS[ ] ▽
    ▽ FIGS;U;DV
[1]   ('ROOT VALUE=';+/EVALUATE T[ROOT;;])
[2]   ('MOST PROMISING=';MINV)
[3]   ('AV. UNDEVELOPED VALUE=';2 RND AV←(+/U/J↑VAL)÷+/
      U←(M1>J↑VAL)^~(1J)∈PAR)
[4]   ('PATH LENGTH=';1+ρROOT PATH L←VAL1MINV)
[5]   8ρ'-'
[6]   ('TREE:')
[7]   ('      DEVELOPED NODES ' ;DV←+/~U;'/' ;J)
[8]   ('      UNDEVELOPED      ' ;+/U;'/' ;J)
[9]   ('      HEIGHT            ' ;HT←[/J↑DEP)
[10]  ('      PENETRANCE        ' ;2 RND PEN←HT÷DV)
[11]  (' DEV APPLICATIONS      ' ;DAPV[ρDAPV])
[12]  FLO←FLOW
[13]  ('FLO(M.P.NODE)= ' ;3 RND FLO[L])
[14]  →0×1~GOAL
[15]  ('DAPV=' ;DAPV)
[16]  ('+/DAPV=' ;+/DAPV)
[17]  ('TIMEV=' ;1 RND TIMEV)
[18]  ('+/TIMEV=' ;1 RND+/TIMEV)
    ▽

      ▽PATH[ ] ▽
    ▽ R←A PATH B;I
[1]   R←(1+DEP[B]-DEP[A])ρ0
[2]   R[I+1]←B
[3]   →PATH1×11=ρR
[4]   R[I+1]←PAR[R[I]]
[5]   →((ρR)>I+I+1)/4
[6]   →(A=R[ρR])/PATH1
[7]   R←10
[8]   PATH1:R←φR
    ▽

```


APPENDIX D

Random Eight-puzzles used (in Experiments 4.1, 4.2 & 4.3).

<i>A</i>			<i>G</i>		
7	5	1	0	3	8
6	4	8	1	2	4
2	3	0	7	6	5
<i>B</i>			<i>H</i>		
2	4	3	3	2	8
6	1	0	4	6	1
7	5	8	0	5	7
<i>C</i>			<i>I</i>		
1	2	7	8	1	7
5	8	4	4	0	2
0	6	3	5	3	6
<i>D</i>			<i>J</i>		
6	1	8	8	5	0
2	5	0	2	3	1
4	3	7	4	6	7
<i>E</i>			<i>K</i>		
8	4	3	6	0	4
5	1	7	5	2	7
6	0	2	3	1	8
<i>F</i>			<i>L</i>		
4	3	7	8	7	2
8	1	2	4	3	0
5	0	6	6	1	5

B29912